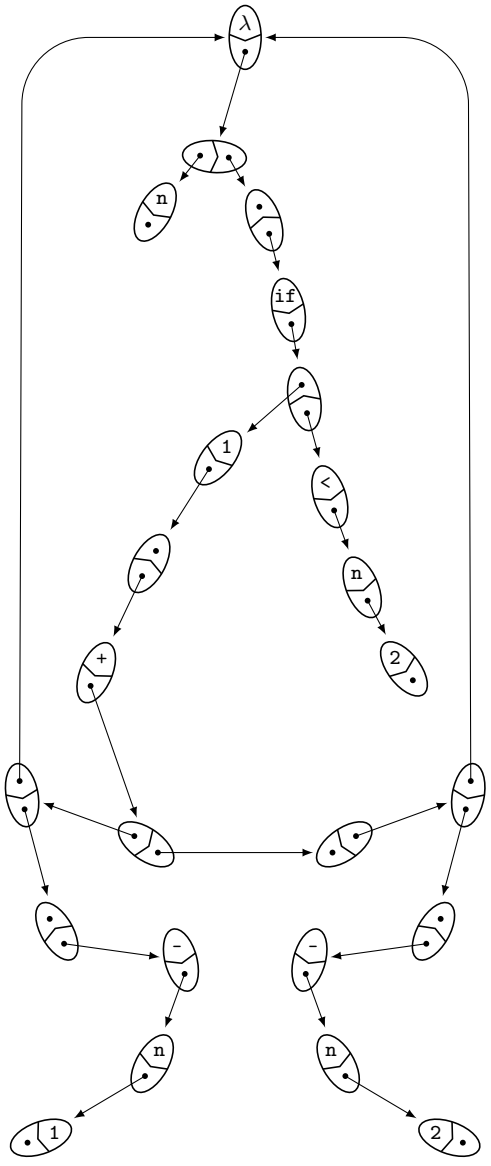
Интерпретация Лиспа и Scheme



Кристиан Кеннек

Данная книга является переводом книги «Les Langages Lisp» (более известной как «Lisp in Small Pieces») Кристиáна Кеннéка (Christian Queinnec), профессора Университета Пьера и Марии Кюри.

Коммит Git: 6f9587036f5e36bf065a0ff4faa5a831fc99da1d.
Дата коммита: 2013-12-07 17:19:55 +0200.
Сборка 7 декабря 2013 г.

Оглавление

К читателю

Несмотря на то, что литературы на тему Лиспа достаточно и она легкодоступна, эта книга имеет свою нишу. Программирование требует понимания фундаментальных основ языка; для Лиспа и Scheme ими являются нетривиальные вещи вроде функций высшего порядка, объектов, продолжений и тому подобного. Их незнание и непонимание преграждает вам путь в будущее, так как то, что ещё сегодня считалось сложным, завтра уже станет нормой для образованного человека.

Для объяснения природы данных сущностей, их происхождения и разновидностей нам придётся серьёзно углубиться в детали. Ходит поговорка, что лисперы знают ценность всего, но не ведают цены. Данная книга также направлена на сокращение этой пропасти в понимании языка с помощью детального изучения его семантики, а также реализации различных возможностей Лиспа, которые были изобретены за его более чем тридцатилетнюю историю.

Лисп — это приятный язык, на котором многие фундаментальные и нетривиальные вещи выражаются простым образом. Вместе с ML, своим строго типизированным собратом, (почти) лишённым побочных эффектов, он является типичным представителем семейства аппликативных языков программирования. Изучение концепций, на которых это семейство основано, без сомнения будет полезно для студентов и учёных-информатиков наших и будущих лет. Основанные на идее *функции*, — идее, которая веками оттачивалась и уточнялась математикой, — аппликативные языки присутствуют практически везде, где присутствуют вычисления, проявляясь в различных формах: начиная перенаправлением потоков в Un⋆x, заканчивая языком расширений редактора Emacs и многими другими скриптовыми языками. Использование таких инструментов без понимания основного их механизма — комбинации — подобно попыткам выразить мысль с помощью отдельных слов вместо цельного предложения. Для выживания может быть достаточно нескольких заученных фраз, но для полноценной жизни требуется вся мощь языка.

Аудитория

Книга предназначена для широкой аудитории специалистов:

- для выпускников вузов и студентов, которые изучают приёмы реализации языков программирования; аппликативных или нет, интерпретацию или компиляцию — не важно;

- для программистов на Лиспе или Scheme, желающих чётче понимать нюансы и стоимость используемых ими конструкций, дабы писать более эффективные и переносимые программы;
- для всех любителей аппликативных языков, которые найдут в этой книге множество интересных размышлений на свою любимую тему.

Философия

Данная книга основана на курсе лекций, читаемом в магистратуре Университета Пьера и Марии Кюри; некоторые части курса также преподаются в Политехнической школе.

Темы, рассматриваемые здесь, обычно следуют за вводным курсом аппликативных языков вроде Лиспа, Scheme или ML, так как подобные курсы чаще всего заканчиваются детальным разбором рассматриваемого языка. Цель этой книги — как можно шире покрыть тему семантики аппликативных языков и разработки их интерпретаторов и компиляторов. Здесь приведено двенадцать интерпретаторов и два компилятора (в байт-код и в Си). Не обходится стороной и объектно-ориентированная модель (рассматриваемая на примере MEROON). Также, в отличие от многих других книг, эта не пренебрегает такими существенными для семейства Лиспов вещами как рефлексия, интроспекция, динамическая кодогенерация и, конечно же, макросы.

Отчасти эта книга вдохновлена двумя работами: «Anatomy of Lisp» [All78], рассматривающей подходы к реализации Лиспа в семидесятых годах, и «Operating System Design: The XINU Approach» [Com84], где приводится весь необходимый код без сокрытия любых деталей работы операционной системы, что полностью убеждает читателя в верности изложения.

В таком же духе — точности, а не лаконичности — написана и эта книга, главным вопросом которой есть семантика аппликативных языков в общем и Scheme в частности. Исследуя множество реализаций, рассматривая их различные аспекты, мы узнаем с максимальной точностью, как строится любая подобная система. Мы рассмотрим большую часть проблемных вопросов, вызывающих расколы в сообществе; каждая из этих проблем будет изучена, варианты её решения — реализованы, сравнены и проанализированы. «Я расскажу всё, что знаю», чтобы вы, читатель, никогда не зашли в тупик из-за недостатка информации, и, более того, имея такой фундамент знаний, могли самостоятельно экспериментировать с рассматриваемыми концепциями.

Именно поэтому вы можете получить в электронном виде полный код всех программ, приведённых в этой книге (подробности на странице 17).

Структура

Книга разделена на две части. Первая часть начинается реализацией наивного интерпретатора Лиспа и рассматривает в основном семантику Scheme. Здесь нам требуется точность повествования, поэтому мы будем раз за разом уточнять и переопределять различными способами пространства имён (Lisp_1, Lisp_2 и т. д.), продолжения (и связанные с ними управляющие конструкции), присваивание и изменяемые структуры данных. Мы заметим, что по мере того, как определяемый язык обрастает возможностями, его определение становится всё более простым, приближаясь к λ-исчислению. Полученное таким образом описание языка мы превратим в его денотационный, строго математический эквивалент.

Более шести лет практики преподавания убедили меня в том, что именно такой подход постепенного уточнения языка необходим для мягкого знакомства с темой исследования языков вообще и денотационной семантикой вычислений в частности — темой, которую мы не можем себе позволить обойти стороной.

Вторая часть книги следует иным путём. Преследуя цель сделать наивную реализацию денотационного интерпретатора более эффективной, мы коснёмся темы ускорения интерпретации (заранее вычисляя неизменные величины), а потом реализуем эту предварительную обработку (с помощью прекомпиляции) для нашего компилятора в байт-код. В этой части подготовка программы к исполнению и собственно исполнение чётко отделены, поэтому здесь будут рассматриваться такие темы как динамические вычисления (`eval`), рефлексия (окружения как объекты первого класса, самоинтерпретация, «башня» интерпретаторов), семантика макросов. Далее мы реализуем транслятор Scheme в код на языке Си.

Завершается книга реализацией объектно-ориентированной системы, которая существенно поможет нам в реализации некоторых интерпретаторов и компиляторов.

Как известно, повторенье — мать ученья. Все приведённые интерпретаторы намеренно написаны в различных стилях: наивном, объектно-ориентированном, основанном на замыканиях, денотационном и т. д. Это позволит рассмотреть множество приёмов, используемых при реализации аппликативных языков. Также это подтолкнёт вас на размышления о различиях между ними. Понимание этих различий (см. таблицу 1 с подсказками) является истинным пониманием языка и его реализаций. Лисп — это не одна из таких реализаций, это *семейство* диалектов, каждый из которых имеет свой уникальный набор черт, которые мы будем рассматривать.

Главы более-менее независимы, занимают примерно по 40 страниц; каждая глава имеет список упражнений, ответы к которым можно найти в конце книги. Список литературы содержит не только исторически важные книги, позволяющие отследить развитие Лиспа с 1960 года, но и современные труды.

Глава	Прототип
1	`(eval exp env)`
2	`(eval exp env fenv)`
	`(eval exp env fenv denv)`
	`(eval exp env denv)`
3	`(eval exp env cont)`
4	`(eval e r s k)`
5	`((meaning e) r s k)`
6	`((meaning e r) sr k)`
	`((meaning e r tail?) k)`
	`((meaning e r tail?))`
7	`(run (meaning e r tail?))`
10	`(->C (meaning e r))`

Таблица 1. Прототипы интерпретаторов и компиляторов.

Предварительные знания

Хоть я и надеюсь, что книга будет увлекательной и содержательной, но она не обязательно будет лёгкой для чтения. Некоторые описанные здесь вещи можно постичь, только если прикладывать усилия, соответствующие их сложности. Говоря языком куртуазных романов, некоторые предметы воздыханий открывают свою истинную красоту и обаяние только тогда, когда мы учтиво, но непреклонно штурмуем их; если их богатый и непростой внутренний мир не будет под постоянной осадой, они так и останутся неприступными.

Изучение сущности языков программирования требует владения инструментами вроде λ-исчисления и денотационной семантики. Хотя повествование и будет мягко, последовательно и логично переходить от одной темы к следующей, это не сможет избавить вас ото всех необходимых усилий.

Вам потребуются некоторые предварительные знания о Лиспе или Scheme; в частности, знание примерно тридцати базовых функций и умение понимать рекурсию без чрезмерного умственного напряжения. Основным языком этой книги выбран Scheme (его краткий обзор можно найти на странице 15), а также его объектно-ориентированное расширение Meroon. Данное расширение поможет нам в рассмотрении некоторых проблем представления и реализации структур данных.

Все приведённые в книге программы были протестированы и действительно работают в интерпретаторе Scheme. А для тех, кто усвоит материал этой книги, не будет составлять особого труда портировать их куда угодно!

Благодарности

Я должен поблагодарить организации, которые обеспечили меня оборудованием (Apple Mac SE/30, затем Sony NEWS 3260, впоследствии разнообразными PC и PowerBook) и вообще сделали эту книгу возможной: Политехническую школу, Государственный институт исследований в области информатики и автоматики (INRIA), Национальный центр научных исследований (CNRS).

Также я хотел бы поблагодарить тех, кто помогал мне всем, чем мог, в создании этой книги. В особом долгу я перед Софи Англад, Жози Бирон, Кэтлин Коллэвей, Жеромом Шейёксом, Жаном-Мари Жеффруа, Кристианом Жюльеном, Жан-Жаком Лакрампом, Мишелем Леметром, Люком Моро, Жаном-Франсуа Перро, Дэниелом Риббенсом, Бернардом Серпеттом, Мануэлем Серрано, Пьером Ве, а также перед моей музой, Клэр Н.*

Конечно же, все ошибки, которые, к сожалению, неизбежно присутствуют в тексте, являются моими собственными.

Нотация

Фрагменты программ будут набраны `таким шрифтом, который несомненно напомнит вам о старых добрых печатных машинках`. Некоторые слова в коде также будут набраны *курсивом* для обозначения понятий, подразумеваемых на месте этих слов.

Знак → читается: «имеет значение», а знак ≡ обозначает эквивалентность, «имеет то же значение, что и». При разборе вычисления выражений после вертикальной черты мы будем записывать окружение, в котором проводятся вычисления. Вот пример, иллюстрирующий эти соглашения:

```
(let ((a (+ b 1)))
  (let ((f (lambda () a)))
    (foo (f) a) ) )|
                   | b → 3
                   | foo ≡ cons

≡ (let ((f (lambda () a))) (foo (f) a))|
                                       | a → 4
                                       | b → 3
                                       | foo ≡ cons
                                       | f ≡ (lambda () a)|
                                                           | a → 4
```

*Sophie Anglade, Josy Byron, Kathleen Callaway, Jérôme Chaillox, Jean-Marie Geffroy, Christian Jullien, Jean-Jacques Lacrampe, Michel Lemaître, Luc Moreau, Jean-François Perrot, Daniel Ribbens, Bernard Serpette, Manuel Serrano, Pierre Weis, Claire N. — *Прим. перев.*

```
≡ (foo (f) a)|
             |a → 4
             |b → 3
             |foo ≡ cons
             |f ≡ (lambda () a)|
                                |a → 4
→ (4 . 4)
```

Все имена переменных и сообщения об ошибках в приводимых программах мы будем записывать на английском — «родном языке» Scheme.

Мы будем использовать несколько нестандартных функций вроде `gensym`, которая генерирует символы, гарантированно не встречавшиеся ранее в тексте программы. В десятой главе также будут применяться функции `format` и `pp` для форматированного вывода (pretty-printing). Эти функции есть в большинстве реализаций Лиспа и Scheme.

Некоторые выражения имеют смысл только для какого-то из диалектов Лиспа вроде Common Lisp, Dylan, EuLisp, ISLisp, Le_Lisp,[1] Scheme и т. д. В этом случае мы будем писать рядом название диалекта:

```
(defdynamic fooncall                                    ISLisp
  (lambda (one :rest others)
    (funcall one others) ) )
```

Дабы было легче ориентироваться в этой книге, мы будем использовать обозначение [**см. стр.**] для перекрёстных ссылок на страницы. Похожая нотация будет использоваться при необходимости указать на упражнение: [**см. упр.**]. Также в книге есть предметный указатель со ссылками на все определяемые функции. [**см. стр.** 587]

Краткий обзор Scheme

Для изучения Scheme существует множество отличных книг, вроде [AS85], [Dyb87], [SF89]. Мы же будем опираться на спецификацию, описанную в документе «Revised revised revised revised revised Report on Scheme», название которого часто сокращают до R^5RS. [KCR98]

Сейчас мы лишь набросаем основные характерные черты этого диалекта; те черты, которые потом будут подробно проанализированы по мере улучшения понимания языка.

В Scheme можно использовать символы, знаки,* строки, списки, числа, логические значения, векторы, порты и функции (или процедуры, как их принято называть в Scheme).

[1] Le_Lisp является торговой маркой INRIA.

* Если возможны разночтения, то слово *знак* будет использоваться в смысле «печатный символ» (character), а слово *символ* — в привычном для Лиспа значении (symbol). — *Прим. перев.*

Каждый из этих типов данных имеет соответствующий предикат: `symbol?`, `char?`, `string?`, `pair?`, `number?`, `boolean?`, `vector?`, `procedure?`.

Помимо них в наличии есть процедуры-аксессоры и модификаторы для тех типов, где это имеет смысл: `string-ref`, `string-set!`, `vector-ref` и `vector-set!`.

Для списков они называются `car`, `cdr`, `set-car!` и `set-cdr!`.

Функции `car` и `cdr` могут комбинироваться. Например, для доступа ко второму элементу списка используется `cadr`.

Все значения этих типов могут быть непосредственно записаны в программе. С символами и числами всё очевидно. Перед знаками пишется префикс `#\`, например: `#\Z`, `#\+`, `#\space`. Строки окружаются `"кавычками"`, списки — `(круглыми скобками)`. Логические значения записываются как `#t` и `#f` соответственно. Для записи векторов используется синтаксис `#(do re mi)`. Естественно, такие значения могут быть построены и динамически с помощью `cons`, `list`, `string`, `make-string`, `vector`, `make-vector`. Также в наличии есть функции приведения типов вроде `string->symbol` и `int->char`.

Ввод-вывод обеспечивают следующие функции: `read` читает вводимые выражения, `display` выводит их на экран, а `newline` переходит на следующую строку.

Программы на Scheme представляются так называемыми *формами*.

Форма `begin` позволяет сгруппировать формы и вычислить их последовательно; например, `(begin (display 1) (display 2) (newline))`.

Есть несколько форм ветвления. Простейшей из них является *if–then–else*, которая на Scheme так и записывается: `(if` *условие тогда иначе*`)`. Если вариантов больше двух, то для этого случая в Scheme есть формы `cond` и `case`. Форма `cond` содержит список утверждений, каждое из которых начинается с условия — выражения, возвращающего логическое значение, — за которым располагается последовательность других форм (следствие). Она последовательно вычисляет условия утверждений до тех пор, пока одно из них не вернёт истину (а точнее: не ложь, не `#f`); затем вычисляется следствие данного утверждения, и результат его вычисления становится результатом всей формы `cond`. Вот пример использования этой формы, который заодно показывает ключевое слово `else`:

```
(cond ((eq? x 'flip) 'flop)
      ((eq? x 'flop) 'flip)
      (else (list x "neither flip nor flop")) )
```

Форма `case` похожа на `cond`, но она принимает первым параметром форму, на основе значения которой производится выбор между вариантами. Каждый из вариантов в начале содержит список значений, которые подходят для него. Как только найден подходящий вариант, он вычисляется и этот результат становится результатом всей формы `case`. Аналогично, в конце может стоять универсальный вариант `else`. Вот так можно переписать предыдущий пример с помощью `case`:

```
(case x
  ((flip) 'flop)
  ((flop) 'flip)
  (else (list x "neither flip nor flop")) )
```

Функции определяются формой `lambda`. За словом `lambda` следует список аргументов, а после него — последовательность выражений, которые описывают собственно вычисление функции. Формы `let`, `let*` и `letrec` определяют локальные переменные (они отличаются тонкостями вычисления начальных значений определяемых переменных). Значения переменных в дальнейшем можно изменять с помощью формы `set!`. Для записи литералов используется форма `quote`.

С помощью формы `define` можно назначить имя любому значению. У неё есть особые возможности, которые мы будем использовать. В частности, возможность использовать её как подобие `let`, а также вариант синтаксиса этой формы, позволяющий удобнее определять функции. Вот, что имеется ввиду:

```
(define (rev l)
  (define nil '())
  (define (reverse l r)
    (if (pair? l) (reverse (cdr l) (cons (car l) r)) r))
  (reverse l nil) )
```

Без синтаксического сахара этот пример выглядит так:

```
(define rev
  (lambda (l)
    (letrec ((reverse (lambda (l r)
                        (if (pair? l) (reverse (cdr l)
                                               (cons (car l) r))
                            r) )))
      (reverse l '()) ) ) )
```

На этом мы заканчиваем наш краткий обзор Scheme.

Исходный код

Программы (интерпретируемые и скомпилированные), приведённые в этой книге, объектную систему и тесты для них можно забрать по следующему адресу:

http://pagesperso-systeme.lip6.fr/Christian.Queinnec/Books/LiSP-2ndEdition-2006Dec11.tgz

Электронный адрес автора книги: Christian.Queinnec@lip6.fr

Рекомендуемая литература

Так как подразумевается, что вы уже знаете Scheme, мы будем ссылаться на традиционные [AS85, SF89].

Чтобы получить от книги больше, имеет смысл поглядывать в другие руководства: Common Lisp [Ste90], Dylan [App92b], EuLisp [PE92], ISLisp [ISO94], Le_Lisp [CDD+91], Oaklisp [LP88], Scheme [KCR98], T [RAM84], Talk [ILO94].

Наконец, для лучшего понимания языков программирования в целом будет полезной книга [BG94].*

* Кроме того, лично я хотел бы посоветовать замечательную книгу *Franklyn Turbak and David Gifford with Mark A. Sheldon.* Design Concepts in Programming Languages. — The MIT Press, 2008. — 1352 p. — *Прим. перев.*

Глава 1

Основы интерпретации

В этой главе описывается базовый интерпретатор, идеи которого проходят красной нитью через большую часть этой книги. Он намеренно сделан простым и более близким к Scheme, чем к Лиспу; это позволит нам в дальнейшем описывать Лисп в терминах Scheme. Мы коснёмся следующих тем в этой вводной главе: сути интерпретации; известной пары функций `eval` и `apply`; ожидаемых свойств окружений и функций. Короче говоря, мы начнём рассматривать здесь то, что будем изучать подробнее в следующих главах, надеясь, что вас не отпугнёт пропасть незнания по обе стороны моста, которым мы пойдём.

Интерпретатор и его варианты будут написаны на Scheme без использования каких-либо существенных особенностей данного диалекта.

В книгах по Лиспу редко когда отказываются от нарциссического соблазна описать Лисп с помощью Лиспа. Начало традиции положило первое руководство по Lisp 1.5 [MAE+62], и впоследствии такой подход широко распространился. Вот лишь малая часть существующих примеров: [Rib69], [Gre77], [Que82], [Cay83], [Cha80], [SJ93], [Rey72], [Gor75], [SS75], [All78], [McC78b], [Lak80], [Hen80], [BM82], [Cli84], [FW84], [dRS84], [AS85], [R3R86], [Mas86], [Dyb87], [WH89], [Kes88], [LF88], [Dil88], [Kam90].

Эти интерпретаторы довольно сильно разнятся, как языками, которые они реализуют и используют для реализации, так и, что более важно, целями, которые они преследуют. Например, интерпретатор из [Lak80] показывает, как объекты и концепции компьютерной графики естественным образом реализуются на Лиспе; а интерпретатор, описываемый в [BM82], создан для явного замера сложности интерпретируемых программ.

Язык, *используемый* для реализации, тоже играет немалую роль. Если в нём есть присваивание и доступ к памяти (`set-car!` и `set-cdr!`), это даёт бóльшую свободу и делает исходный код интерпретатора более компактным. Мы получаем возможность описать язык в терминах, которые близки к машинным инструкциям. И, как и с машинными инструкциями, несмотря на то, что такое описание может быть непросто составить, не возникает никаких сомнений в том, что именно делает каждая конкретная строка. Даже если подобное описание занимает больше места, чем высокоуровневое, оно даёт более

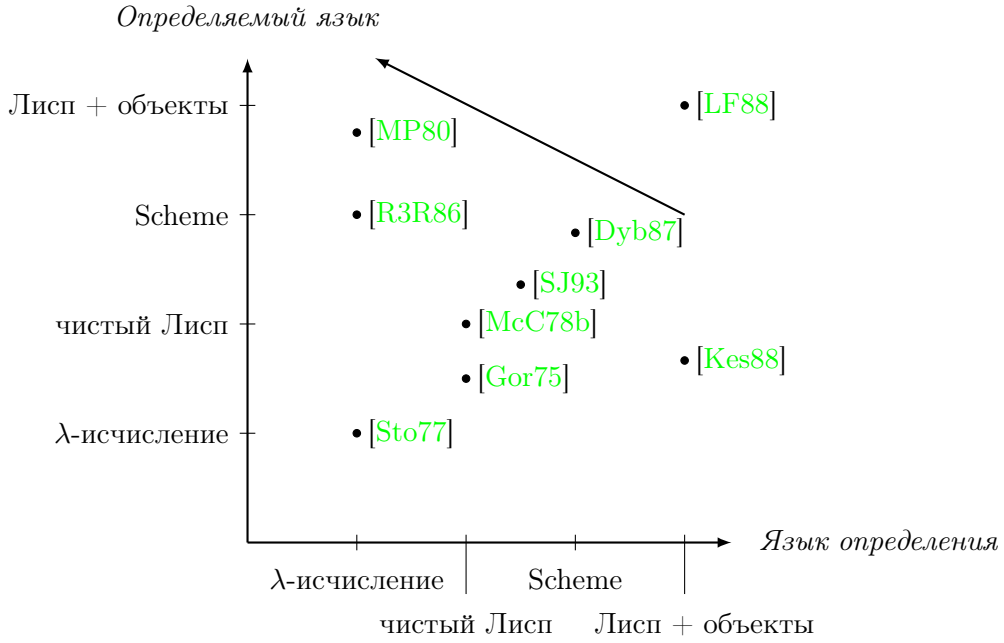


Рис. 1.1. Уровни сложности.

точное понимание смысла происходящего при интерпретации — для нас важно именно это.

На рисунке 1.1 показано сравнение уровней сложности определяющего (по оси x) и определяемого (по оси y) языков для некоторых из интерпретаторов. Здесь хорошо виден ход развития наших знаний со временем: всё более сложные проблемы решаются с использованием всё более ограниченных возможностей. Эта книга соответствует вектору, который начинается использованием высокоуровневого Лиспа для реализации Scheme, а заканчивается реализацией высокоуровневого Лиспа с помощью одного лишь λ-исчисления.

1.1. Вычисления

Важнейшая часть интерпретатора Лиспа находится в одной функции, вокруг которой крутится все остальное. Эта функция, называемая `eval`, принимает на вход программу, а на выходе даёт результат её исполнения. Явное наличие исполнителя кода отнюдь не случайно, а, наоборот, является характерной и намеренно реализованной чертой Лиспа.

Язык программирования называется *универсальным*, если он не уступает в выразительных возможностях машине Тьюринга. Так как машина Тьюринга довольно примитивна, то несложно разработать язык, который бы удовлетворял этому условию; действительно, сложнее будет придумать язык, который был бы полезным, но при этом не был бы полным по Тьюрингу.

В соответствии с тезисом Чёрча, любая вычислимая функция может быть записана на любом тьюринг-полном языке. Интерпретатор Лиспа можно представить как такую функцию, которая принимает программы и возвращает результаты их исполнения. Так что если такая функция вообще существует, её можно записать на любом тьюринг-полном языке. Следовательно, функцию-вычислитель Лиспа `eval` можно записать в частности на том же самом Лиспе. В этом нет никаких противоречий, точно так же, к примеру, можно реализовать Фортран на Фортране.

Но что делает Лисп уникальным (и оправдывает явное описание `eval`), так это небольшой размер кода интерпретатора: обычно от одной до двадцати страниц в зависимости от детализации.[1] Это результат желания сделать язык последовательным, с минимальным количеством исключений из правил, и, что самое главное, с простым, но выразительным синтаксисом.

Сам факт существования `eval`, а также возможность её описания на Лиспе имеют несколько интересных следствий.

- Можно изучить Лисп или прочитав руководство (в котором описываются все доступные функции), или изучив собственно функцию `eval`. Второй подход сложен тем, что надо уже знать Лисп для того, чтобы понять описание `eval`; но ведь знание Лиспа по идее должно быть *следствием* изучения `eval`, нежели *предпосылкой* для него. На самом деле, достаточно знать лишь ту часть Лиспа, которая используется для описания `eval`. Кроме того, язык, определяемый одной `eval`, не является всем Лиспом: он есть лишь сутью языка, в нём реализованы только специальные формы и немного примитивных функций.

  Тем не менее, в возможности изучать язык двумя разными, но всё же связанными путями лежит несомненный плюс Лиспа.

- Тот факт, что `eval` написана на Лиспе, значит также и то, что среда разработки является составной частью языка и не требует значительных накладных расходов. Под средой разработки понимаются такие вещи как отладчик, трассировщик или возможность обратного хода [Lie87]. Практически, реализация таких инструментов — это лишь доработка `eval`, к примеру, чтобы она выводила сообщения при вызове функций, приостанавливала вычисления в интересующих местах и так далее.

  Долгое время среда разработки с такими возможностями была уникальной для Лиспа. Но и сегодня то, что `eval` может быть описана на самом Лиспе, даёт возможность легко экспериментировать с новыми вариантами реализации вычислений или отладки.

- Наконец, сама по себе `eval` способна быть инструментом программирования. Достаточно спорным инструментом, так как использование `eval` требует присутствия в памяти целого интерпретатора или компилятора

[1] Интерпретатор, описываемый в этой главе, занимает около 150 строк.

во время исполнения кода; но ещё более серьёзной проблемой является невозможность применения в таком случае некоторых оптимизаций. Другими словами, использование `eval` имеет свою цену. В некоторых случаях её использование полностью оправдано, к примеру, когда Лисп используется для описания и реализации метаязыков.

Кроме ощутимой стоимости использования, семантика `eval` часто неоднозначна. Именно поэтому она вообще не входила в стандарт до ревизии R^{5}RS [CR91b, KCR98]. [**см. стр. 325**]

1.2. Базовый вычислитель

Будем различать в программе *свободные* и *связанные переменные*. Переменная свободна, если ни одна связывающая форма (вроде `let` или `lambda`) не связывает её с каким-либо значением. В противном случае она называется связанной. Соответственно, свободные переменные могут иметь любое значение, и нельзя определённо сказать, какое именно, не выходя за рамки функции. Структура данных, которая связывает переменные с их значениями, называется *окружением*. Таким образом, функция `evaluate`[2] имеет два аргумента: программу и окружение, в котором она должна быть исполнена:

```
(define (evaluate exp env) ... )
```

1.3. Вычисляем атомы

Одна из ключевых особенностей Лиспа состоит в том, что программы записываются теми же конструкциями, что и остальные данные. Но так как любая запись подразумевает определённые условности, поговорим о таких условностях для записи программ. Главные соглашения: переменные записываются символами (своими именами), а вызовы функций — списками, где первый элемент — это вызываемая функция, а остальные — её аргументы.

Как и любой другой компилятор, `evaluate` начинает свою работу с синтаксического анализа переданного ей выражения, чтобы выяснить, что именно оно означает. В этом смысле, название раздела не совсем правильное: мы рассматриваем не буквально вычисление атомов, а интерпретацию программ, состоящих из атомов. Сейчас важно отличать саму программу от её представления (письмо от листа бумаги, на котором оно написано). Функция `evaluate` работает с представлением программ, по нему она узнаёт предписываемые программой действия и выполняет их.

[2] Как вы знаете, «родная» функция-вычислитель Scheme называется `eval`. Чтобы их не путать, нашу функцию мы назовём `evaluate`. По тем же причинам местный аналог `apply` будет называться `invoke`. Меньше перекрытий имён — меньше проблем.

```
(define (evaluate exp env)
  (if (atom? exp)   ; (atom? exp) ≡ (not (pair? exp))
      ...
      (case (car exp)
        ...
        (else ...) ) ) )
```

Если выражение не является списком, то это скорее всего символ, или число, или какая-нибудь строка. Если это всё же символ, то он представляет *переменную*, а её значение хранится в окружении:

```
(define (evaluate exp env)
  (if (atom? exp)
      (if (symbol? exp) (lookup exp env) exp)
      (case (car exp)
        ...
        (else ...) ) ) )
```

Функция `lookup` (которая рассматривается подробнее на странице 32) знает, как отыскать значение переменной в окружении. Вот её прототип:

`(lookup` *переменная окружение*`)` → *значение*

Как видим, здесь происходит неявное преобразование символа в переменную. Если быть более дотошным, тогда вместо `(lookup exp env)` надо записать что-то вроде:

```
... (lookup (symbol->variable exp) env) ...
```

В таком случае мы явно говорим, что символ `exp`, чьим значением является имя переменной, должен быть превращён в переменную. Также это подчёркивает тот факт, что функция `symbol->variable`[3] вовсе не переводит `exp` сам в себя; она превращает синтаксическую сущность (символ) в семантическую (переменную). В действительности, переменные — это лишь воображаемые объекты, которым язык и программист дали какие-то имена, и которые ради удобства используются в форме имён. Способ представления имён также выбран из соображений удобства, так как Лисп имеет базовый тип символов. В данном случае `symbol->variable` ничего не делает, хотя на самом деле могла бы применяться какая-нибудь другая форма записи имени переменной, например: строка, состоящая из знака доллара и имени переменной; в этом случае, конечно же, `symbol->variable` будет сложнее.

Если бы переменные действительно были лишь воображаемыми, то `lookup` не знала бы как обрабатывать свой первый аргумент, так как она ожидает нечто «материальное». Так что нам надо преобразовать переменную в её программное представление, какой-нибудь уникальный ключ, по которому `lookup`

[3] Лично я не люблю называть функции приведения типов x`->`y, потому что так сложнее понимать цепочки преобразований. Запись `(`y`->`z`(`x`->`y `...))` не так очевидна, как `(`z`<-`y`(`y`<-`x `...))`. Хотя, с другой стороны, одиночная запись z`<-`y не так легко читается, как y`->`z. Приходится выбирать.

сможет отыскать переменную в окружении. Так что ещё точнее было бы записать:

```
... (lookup (variable->key (symbol->variable exp)) env) ...
```

Однако, врождённая лень лисперов настаивает на использовании символов для ключей. Так что `variable->key` — это лишь обратная функция к `symbol->variable`, и их последовательное применение никак не изменяет `exp`.

Если выражение атомарное (то есть не является списком) и не является символом, то соблазнительно его считать представлением какой-нибудь константы с соответствующим значением. Такое поведение называется механизмом *автоцитирования*. Автоцитируемый объект не требует явного цитирования и имеет собственное значение. За примерами можно обратиться к [Cha96].

Но является ли такое поведение правильным? Во-первых, не всегда атомарные объекты обозначают сами себя. Например, строка `"a?b:c"` могла бы означать вызов компилятора Си, затем вызов получившейся программы и подстановку возвращаемого ей значения вместо этой строки.

С другими объектами (вроде функций) вообще не понятно, как именно их *вычислять*. К примеру, ясно, что значением переменной `car` является функция, возвращающая левый элемент пары, но что является значением самой функции *car*? Чаще всего попытки вычисления значения функции считаются ошибочными.

Другой пример проблемного значения — пустой список `()`. Судя по тому, что это список, он должен означать вызов функции. Вот только в нём нет ни аргументов, ни самой функции. Такая запись в Scheme запрещена и является синтаксической ошибкой.

Поэтому необходимо очень аккуратно анализировать программу и автоцитировать только те данные, для которых это явно стоить делать, например: числа, строки и знаки. [**см. стр. 26**] Так что мы записываем следующее:

```
(define (evaluate exp env)
  (if (atom? exp)
      (cond ((symbol? exp) (lookup exp env))
            ((or (number? exp) (string? exp) (char? exp)
                 (boolean? exp) (vector? exp) )
             exp )
            (else (wrong "Cannot evaluate" exp)) )
      ... ) )
```

В этом фрагменте кода виден первый случай, когда могут возникнуть ошибки. Большая часть лисп-систем имеет свои собственные механизмы обработки исключительных ситуаций, которые непросто сделать переносимыми. В случае ошибки мы вызываем `wrong`[4] и передаём ей первым аргументом строку. В строке находится текстовое описание ошибки, а следующие за ней аргументы несут дополнительную информацию о том, что вызвало про-

[4] Заметьте, не «функцию `wrong`». Варианты её реализации подробнее рассматриваются в разделе 7.9.

блему. Системы с зачаточным механизмом обработки ошибок обычно в случае проблем выдают какие-то непонятные надписи вроде `"Bus error: core dumped"` и умирают. Другие останавливают текущие вычисления и возвращаются к диалоговому режиму. А третьи могут связывать с вычисляемым выражением специальный обработчик исключений, который перехватит брошенный объект, описывающий ошибку, и будет уже по нему решать, что делать дальше. [**см. стр. 307**] В некоторых случаях даже реализуется подобие экспертной системы, которая анализирует ошибку, вызвавший её код и выдаёт пользователю варианты её исправления. В общем, сложно сказать однозначно, что следует делать в случае ошибки.

1.4. Вычисляем формы

Каждый язык имеет некоторое количество «неприкасаемых» синтаксических конструкций: их нельзя адекватно переопределить и вообще не стоит трогать. В Лиспе такие конструкции называются *специальными формами*. Они представляются списками, где первый элемент — это определённый символ, принадлежащий множеству *специальных операторов* (или синтаксических ключевых слов, как их называют в Scheme).

Конкретный диалект Лиспа характеризуется набором специальных форм и библиотекой примитивных функций (эти функции нельзя определить на самом диалекте, так как они тесно связаны с реализацией; например, для Scheme это `call/cc`).

В некотором понимании, Лисп является лишь прикладным λ-исчислением вместе с расширяющим его набором специальных форм. Специфика каждого конкретного диалекта Лиспа лежит только в этом наборе. В Scheme используется минимальный набор специальных операторов (`quote`, `if`, `set!` и `lambda`), тогда как Common Lisp (CLtL2 [Ste90]) определяет более тридцати, описывая таким образом случаи, когда может быть сгенерирован высокоэффективный машинный код.

Так как специальные формы записываются буквально, то их синтаксический анализ прост, хватит одного `case`-выражения: надо лишь смотреть на первый элемент списка. Если форма начинается не со специального оператора, то она означает применение функции. В данный момент мы ограничимся лишь небольшим подмножеством специальных форм: `quote`, `if`, `begin`, `set!` и `lambda`. (В следующих главах мы введём другие, более специализированные формы.)

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) ) e)
            (else (wrong "Cannot evaluate" e)) )
```

```
(case (car e)
  ((quote)  (cadr e))
  ((if)     (if (evaluate (cadr e) env)
                (evaluate (caddr e) env)
                (evaluate (cadddr e) env) ))
  ((begin)  (eprogn (cdr e) env))
  ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
  ((lambda) (make-function (cadr e) (cddr e) env))
  (else     (invoke (evaluate (car e) env)
                    (evlis (cdr e) env) )) ) ) )
```

Чтобы упростить определение, синтаксический анализ оставлен минимальным: мы не проверяем, правильно ли записаны цитаты, действительно ли `if` передано три аргумента[5] и так далее. Мы априори считаем интерпретируемые программы синтаксически корректными.

1.4.1. Цитирование

Специальная форма `quote` позволяет записать значение, которое без цитирования было бы спутано с каким-нибудь выражением. Такой механизм необходим, если программы тоже являются типом данных в языке — надо же как-то различать, где записана программа, а где данные. Если бы был выбран другой синтаксис, то эта проблема бы не возникла. К примеру, изначально в Лиспе планировались M-выражения [McC60] для записи действий над данными, а сами данные должно были записываться S-выражениями. Это решило бы проблему разделения кода и данных, но сделало бы невозможными вменяемые макросы — очень полезную вещь для расширения синтаксиса. Как бы то ни было, M-выражения долго не прожили [McC78a], программы и данные стали записываться исключительно S-выражениями. Поэтому в Лиспе есть специальная форма `quote`, служащая для разделения кода и данных.

Суть цитирования состоит в возврате выражения, следующего за ключевым словом, «как есть», без его вычисления. Это чётко видно в следующем фрагменте кода:

```
... (case (car e)
      ((quote) (cadr e)) ... ) ...
```

Также интересен вопрос: а есть ли разница между явным и неявным цитированием, например, между `33` и `'33`, или между[6] `#(fa do sol)` и `'#(fa do sol)`? Если первое сравнение происходит между непосредственными значениями и тут вроде бы всё очевидно, то во втором случае сравниваются составные объекты (хоть для Лиспа они и являются формально атомами). Можно

[5] `if` не обязательно принимает условие и ровно две альтернативы. К примеру, в Scheme и Common Lisp `if` может принимать как два, так и три аргумента; `if` в EuLisp и ISLisp исключительно тернарный; Le_Lisp в случае, если условие ложно, вычисляет остаток формы `if`, обернув его в `begin`.

[6] Напомним: в Scheme запись `#(...)` означает процитированный вектор.

легко придумать несколько возможных трактовок данного выражения. Цитирование возвращает аргумент как значение, но сама запись `#(fa do sol)` могла бы возвращать каждый раз новый вектор из трёх символов. Другими словами, `#(fa do sol)` могла бы быть просто сокращённой записью выражения `(vector 'fa 'do 'sol)`, которое, конечно же, будет означать совсем иное, нежели `'#(fa do sol)`, и тем более `(vector fa do sol)`. Мы вернёмся к этой проблеме позже [**см. стр. 175**], потому что, как видите, не так просто определиться, какой именно смысл придавать цитированию.

1.4.2. Ветвление

Вспомним, как работает условный оператор `if`: эта форма вычисляет свой первый аргумент (*условие*), затем в зависимости от результата вычислений выбирает, возвращать значение второго аргумента (*следствия*) или третьего (*альтернативы*). Эта идея выражается следующим кодом:

```
... (case (car e)
      ((if) (if (evaluate (cadr e) env)
                (evaluate (caddr e) env)
                (evaluate (cadddr e) env) )) ... ) ...
```

Эта программа не совсем точно учитывает представление логических значений. Явно видно, что здесь смешиваются два языка: Scheme (или хотя бы что-то неотличимо похожее на него) и Scheme (или что-то довольно похожее). Причём мы определяем второй в терминах первого. Так что между ними присутствуют примерно те же отношения, что и между Паскалем (на котором написана первая реализация TEX) и самой системой TEX [Knu84]. Соответственно, нет ни единого повода считать способы представления логических значений в этих языках одинаковыми.

Функция `evaluate` возвращает значения определяемого языка. Они априори никак не связаны с логическими значениями языка, используемого для реализации. Следуя соглашению о том, что любой объект, не равный логической *лжи*, должен считаться логической *истиной*, мы запишем:

```
... (case (car e)
      ((if) (if (not (eq? (evaluate (cadr e) env) the-false-value))
                (evaluate (caddr e) env)
                (evaluate (cadddr e) env) )) ... ) ...
```

Здесь считается, что переменная `the-false-value` хранит значение *лжи* определяемого языка, выраженное в терминах определяющего. Вариантов выбора такого представления много, например, сделаем так:

```
(define the-false-value (cons "false" "boolean"))
```

В этом случае всё будет замечательно, так как `eq?` в Scheme сравнивает не значения, а адреса, и, соответственно, любое другое значение никак нельзя будет спутать со значением переменной `the-false-value`.

Вопрос представления логических значений вовсе не тривиален. В истории Лиспа полно споров на тему различий между булевым значением *ложь*, пустым списком `()` и символом `NIL`. Наиболее чёткая позиция по этому вопросу: *ложь* это не `()` (в конце концов, это всего лишь пустой список), и они оба тем более не имеют никакого отношения к символу, составленному из букв `N`, `I` и `L`.

Такую позицию занимает и Scheme; об этом всё же смогли договориться за пару недель до принятия стандарта IEEE [IEE91].

Так что в Scheme теперь всё хорошо (разве что `()` по-английски всё ещё читается как *nil*!). В изначальном Лиспе *ложь*, `()` и `NIL` — это один и тот же символ. В Le_Lisp `NIL` это переменная со значением `()`, а пустой список (вместе с пустым символом `||`) используется в качестве *лжи*.

1.4.3. Последовательность

Существует специальная форма, позволяющая вычислить группу форм последовательно и в определённом порядке. Как и старые добрые блоки `begin ... end` из семейства языков Алгола, в Scheme эта форма называется `begin`; в других Лиспах она обычно зовётся `progn` — обобщенная версия `prog1`, `prog2` и т. д. Собственно организацию последовательности мы перепоручаем функции `eprogn`.

```
... (case (car e)
      ((begin) (eprogn (cdr e) env)) ... ) ...

(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      '() ) )
```

Такое определение последовательных вычислений не допускает разночтений. Стоит обратить внимание на хвостовую рекурсию при вычислении последней формы последовательности. Вычисления построены так, что последний вызов `evaluate` заменяет собой всю рекурсивную цепочку вызовов `eprogn`. (О хвостовой рекурсии мы позже поговорим подробнее. [**см. стр. 134**])

Ещё одним интересным моментом является то, что́ возвращается при вычислении формы `(begin)`. Сейчас это пустой список. Но почему именно `()`, почему не что-то другое, вроде `:3` или `(^_^)`? Мы выбрали пустой список по привычке, доставшейся в наследство от Лиспа: в любой непонятной ситуации возвращай `nil`. Но в мире, где *ложь*, `nil` и `()` — это совершенно различные вещи, что из них лучше подходит на роль *ничего*? Поэтому пусть в нашем языке вычисление `(begin)` будет возвращать специальное значение `empty-begin`, которое определяется как (почти) случайное число `813` [Leb05].

```
(define (eprogn exps env)
  (if (pair? exps)
      (if (pair? (cdr exps))
          (begin (evaluate (car exps) env)
                 (eprogn (cdr exps) env) )
          (evaluate (car exps) env) )
      empty-begin ) )

(define empty-begin 813)
```

Корни нашей проблемы в том, что `begin` *обязана* вернуть какое-то значение. Как и Scheme, определяемый язык может не придавать какого-либо смысла форме `(begin)`; мы можем или допускать такое написание и возвращать любое удобное значение, или же не допускать и считать ошибкой. Идеальный вариант: вообще не использовать `begin` без аргументов, так как не определено, что именно получится в результате. Некоторые реализации имеют специальный объект: `#<unspecified>`, который возвращается в случае, когда нет ничего более подходящего. Обычно единственное, что с ним можно сделать — это вывести на печать. (Не следует путать этот объект с псевдозначением у неопределённых переменных. [**см. стр. 83**])

Последовательное вычисление выражений бесполезно в чисто функциональных языках (где функции не имеют побочных эффектов). Действительно, какой в таком случае смысл что-то вычислять и не использовать возвращаемые значения? Но иногда в этом может быть смысл. Представим себе игру, написанную на чисто функциональном языке; очевидно, что вычисления занимают какое-то время вне зависимости от того, используются ли их результаты или нет; и нас может интересовать именно этот «побочный эффект» — замедление работы, — а не получаемые результаты. Тогда можно последовательно что-то вычислять, например, чтобы скорость игры была адекватна рефлексам игрока (если только компилятор не посчитает себя самым умным и не удалит «бесполезный» код).

Так как в Scheme есть операции ввода-вывода, которые имеют побочные эффекты, то для нас есть смысл пользоваться формой `begin`, потому как очевидно, что лучше сначала задать вопрос (с помощью `display`), а потом прочитать ответ (с помощью `read`), чем сделать наоборот. Здесь-то как раз и нужно упорядочить вычисления. Но не только `begin` может их упорядочивать. Например, условный оператор может:

`(if` α β β`)` $\equiv$ `(begin` α β`)`

И `lambda` тоже может[7]:

`(begin` α β`)` $\equiv$ `((lambda (`*void*`)` β`)` α`)`

[7] Переменная *void* не должна быть свободной в β. Это условие выполняется, если *void* никогда не встречается в β. Обычно в таком случае используется `gensym`, чтобы получить гарантированно уникальное имя переменной. [**см. упр. 1.11**]

Как видно из этого примера, в Scheme `begin` не является необходимой специальной формой, так как её поведение можно проэмулировать с помощью функций благодаря тому, что при вызове функции её аргументы вычисляются перед исполнением тела (передача аргументов *по значению*).

1.4.4. Присваивание

Как и во многих других языках, в нашем диалекте значения переменных можно менять. Изменение значения переменной называется *присваиванием*. Так как значение переменной надо изменять в её окружении, то мы оставляем эту проблему функции `update!`.[8] Её настоящая суть объясняется позже, на странице 163.

```
... (case (car e)
      ((set!) (update! (cadr e) env
                       (evaluate (caddr e) env))) ... ) ...
```

Присваивание выполняется в два шага: сначала вычисляется новое значение, потом новое значение заменяет старое. Стоит заметить, что обновлённая переменная не является значением данной формы. Мы ещё вернёмся к вопросу о возвращаемом значении операции присваивания. [**см. стр. 142**] Пока только запомните, что стандартом оно не определено.

1.4.5. Абстракция

Функции (также называемые *процедурами* в Scheme) являются результатом вычисления специальной формы `lambda`, чьё имя ссылается на понятие *абстракции* в λ-исчислении. Работу по созданию функции мы поручаем функции `make-function`, которой передаём всё необходимое: список аргументов, тело функции и текущее окружение.

```
... (case (car e)
      ((lambda) (make-function (cadr e) (cddr e) env)) ... ) ...
```

1.4.6. Аппликация

Если первый элемент списка не является специальным оператором, то такой список означает применение функции, которое в λ-исчислении называется *аппликацией* или *комбинацией*. Функция, полученная в результате вычисления первого элемента, применяется к аргументам, которые мы получим, вычислив остальные элементы списка. Эти действия описываются следующим кодом:

[8] В соответствии с принятым в Scheme соглашением, имена функций с побочными эффектами оканчиваются на восклицательный знак.

```
... (case (car e)
      (else (invoke (evaluate (car e) env)
                    (evlis (cdr e) env) )) ) ...
```

Вспомогательная функция `evlis` принимает список выражений и возвращает список соответствующих им значений. Она определяется следующим образом:

```
(define (evlis exps env)
  (if (pair? exps)
      (cons (evaluate (car exps) env)
            (evlis (cdr exps) env) )
      '() ) )
```

Далее работает функция `invoke`, которая применяет свой первый аргумент (функцию; если это не так, она сообщает об ошибке) ко второму (списку её аргументов) и возвращает результат вычислений. В общем, `invoke` похожа на привычную `apply`, разве что требует явного указания окружения. (Далее в разделе 1.6 [**см. стр. 34**] мы рассмотрим подробнее непростые взаимоотношения функций и окружений.)

Ещё немного об `evaluate`

Рассмотренное описание языка является более-менее точным. Мы не разобрали лишь несколько вспомогательных функций: `lookup` и `update!`, отвечающие за окружения, и `make-function` вместе с `invoke`, занимающиеся функциями. Но даже сейчас нам уже многое известно об `evaluate`. Например, наш диалект имеет единое пространство имён, понятие объекта в нём распространяется вообще на всё (как в Lisp_1 [**см. стр. 51**]), в том числе и на функции. Но мы до сих пор не знаем порядок вычисления аргументов.

В нашем случае он зависит от порядка вычисления аргументов у `cons`, которая используется в `evlis`. Но мы легко можем указать любой понравившийся нам порядок, например, слева направо:

```
(define (evlis exps env)
  (if (pair? exps)
      (let ((argument1 (evaluate (car exps) env)))
        (cons argument1 (evlis (cdr exps) env)) )
      '() ) )
```

Без введения новых специальных конструкций[9] мы уточнили поведение определяемого языка. Первая часть книги направлена именно на это: уточнение определяемого языка с помощью всё более ограниченных возможностей, что снижает зависимость описания от языка, используемого для определения.

[9] Как известно, `let` — это всего лишь простой макрос: `(let ((`x π_1`)) `π_2`) ≡ ((lambda (`x`) `π_2`) `π_1`)`.

1.5. Представление окружений

Окружения связывают переменные с их значениями. Обычно в Лиспе подобные связи представляются *ассоциативными списками*, также называемыми *A-списками*. Мы тоже будем представлять окружения как A-списки, связывающие переменные и значения. Для простоты имена переменных будем представлять символами.

Таким образом, функции `lookup` и `update!` определяются элементарно:

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (cdar env)
          (lookup id (cdr env)) )
      (wrong "No such binding" id) ) )
```

Тут мы видим второй тип[10] возможных ошибок, появляющихся при попытке узнать значение неизвестной переменной. Мы опять лишь вызовем `wrong`, чтобы сообщить о проблеме наверх.

Когда компьютеры были большими, а память была маленькой,[11] для переменных часто применялось *автоцитирование*. Если с переменной не было связано какое-либо значение, то этим значением становился символ с именем переменной. Было бы очень обидно видеть, как понятия переменной и символа, которые мы так усердно разделяли, опять смешиваются и перепутываются.

Хотя это несомненно удобно — никогда не вызывать ошибок, но такой идеальный мир имеет большой недостаток: задача программы не в том, чтобы работать без ошибок, а в том, чтобы выполнять то, для чего она предназначена. В этом смысле ошибки играют роль перил: если мы на них натыкаемся, то это значит, мы идём куда-то не туда. Ошибки должны быть обнаружены как можно раньше, чтобы как можно быстрее их исправить. Следовательно, использование автоцитирования — плохое решение, потому что оно скрывает некоторые ошибки, которые могли бы быть исправлены раньше.

Функция `update!` изменяет окружение, так что, скорее всего, тоже может вызвать такую же ошибку: нельзя изменить значение неизвестной переменной. Мы обсудим, следует ли ей так поступать, когда будем говорить о глобальном окружении.

```
(define (update! id env value)
  (if (pair? env)
      (if (eq? (caar env) id)
          (begin (set-cdr! (car env) value)
                 value)
          (update! id (cdr env) value) )
      (wrong "No such binding" id) ) )
```

[10] Первый — это синтаксические ошибки (стр. 24).

[11] Память (вместе с подсистемами ввода-вывода) всё ещё остаётся одной из наиболее дорогих частей компьютера, хоть и постоянно дешевеет.

Возвращаемое значение функции `update!` выбрано с учётом того, что это значение станет значением всего выражения присваивания. В Scheme это значение не определено. Строго говоря, программам не следует полагаться на какое-то значение, но, тем не менее, мы вынуждены выбирать, что именно будем возвращать в нашем случае. Например:

1) только что присвоенное значение (так сделано сейчас);

2) предыдущее значение переменной (могут быть проблемы с инициализацией, первым присваиванием);

3) объект «неопределённое значение», некий `#<UFO>`, используемый исключительно как индикатор неопределённого значения;

4) значение формы с неопределённым значением, вроде `set-cdr!` в Scheme.

Окружения — это составной абстрактный тип данных. Мы уже можем извлекать и изменять их части с помощью соответствующих функций; но ещё надо уметь создавать новые окружения и добавлять в них новые части.

Изначально в окружении ничего нет. Это записывается просто:

```
(define env.init '())
```

(Чуть позже, в разделе 1.6, мы сделаем его не таким необитаемым.)

Когда вызывается функция, для неё создаётся новое окружение, в котором её аргументы связаны со своими фактическими значениями. Функция `extend` расширяет окружение `env` переменными `variables` с соответствующими значениями `values`.

```
(define (extend env variables values)
  (cond ((pair? variables)
         (if (pair? values)
             (cons (cons (car variables) (car values))
                   (extend env (cdr variables) (cdr values)) )
             (wrong "Too few values") ) )
        ((null? variables)
         (if (null? values)
             env
             (wrong "Too many values") ) )
        ((symbol? variables) (cons (cons variables values) env)) ) )
```

Главная сложность состоит в том, что нам надо проанализировать все варианты записи <*списка аргументов*>, какие разрешены в Scheme.[12] Список аргументов может быть представлен не только обычным списком символов,

[12] Некоторые лисп-системы, вроде Common Lisp, поддались соблазну расширить синтаксис перечня аргументов ключевыми словами вроде `&aux`, `&key`, `&rest`. Но это сильно усложняет синтаксический разбор и связывание. Иные системы позволяют даже обобщённое указание аргументов с помощью шаблонов [SJ93].

но и точечным: заканчивающимся не на `()`, а на определённый символ (*точечную переменную*). Более формально список аргументов описывается следующей грамматикой:

$$
\begin{aligned}
&\langle \textit{список аргументов} \rangle ::= \texttt{()} \\
&\qquad\qquad\qquad\qquad \mid \langle \textit{переменная} \rangle \\
&\qquad\qquad\qquad\qquad \mid \texttt{(}\langle \textit{переменная} \rangle \texttt{ . } \langle \textit{список аргументов} \rangle\texttt{)} \\
&\qquad \langle \textit{переменная} \rangle \in \textbf{Символы}
\end{aligned}
$$

Когда мы расширяем окружение, количество значений переменных должно соответствовать количеству их имён. Обычно их поровну, но если список оканчивается на точечную (или *n-арную*) переменную, то она связывается со списком всех оставшихся аргументов. Как бы то ни было, возможны две ошибки: или значений больше, чем переменных, или наоборот.

1.6. Представление функций

Наверное, проще всего представлять функции с помощью функций. Естественно, это не тавтология, и читать следует так: «функции определяемого языка проще всего представлять функциями языка определения». Это сильно упрощает механизм вызова: функция `invoke` должна лишь проверить, действительно ли её первый аргумент является функцией: чем-то, что можно вызвать.

```
(define (invoke fn args)
  (if (procedure? fn) (fn args)
      (wrong "Not a function" fn) ) )
```

Проще некуда. Даже может возникнуть вопрос, зачем вообще нужна отдельная функция, когда весь этот код можно было встроить сразу в `evaluate`. Причина, по которой так сделано, в том, что мы показываем структуру будущих интерпретаторов, а в них `invoke` будет уже не такой простой. Кстати, попробуйте сейчас выполнить упражнения 1.7 и 1.8.

Также у нас появляется новый тип ошибок, возникающих при попытке вызвать невызываемое. Сейчас мы обрабатываем такие ошибки в момент применения функции к уже вычисленным аргументам, но мы могли бы предупреждать пользователя раньше. В таком случае нам необходимо задать порядок вычисления элементов формы вызова функции:

1) вычислить элемент на месте функции;
2) если это не функция, сообщить об ошибке;
3) вычислить аргументы слева направо;
4) сравнить количество аргументов с арностью функции и, если они не совпадают, то сообщить об ошибке.

Вычислять аргументы слева направо кажется логичным для людей, читающих слева направо. Это и запрограммировать легче, в итоге порядок прост и понятен. Сложности возникают только у компилятора, потому что если ему захочется поменять этот порядок (например, чтобы эффективнее использовать регистры процессора), то он будет вынужден доказать, что это не изменит смысла программы.

Конечно, мы могли бы действовать эффективнее, проверяя арность ещё раньше:

1) вычислить элемент на месте функции;

2) если это не функция, сообщить об ошибке, иначе запомнить ожидаемое количество аргументов;

3) вычислять аргументы слева направо до тех пор, пока их количество согласуется с арностью функции, в случае проблем сообщить об ошибке;

4) применить функцию к аргументам.[13]

Стандарт Common Lisp требует, чтобы аргументы вычислялись строго слева направо, но с целью оптимизации позволяет вычислять функциональный элемент списка до или после остальных.

Scheme же не накладывает условий на порядок вычисления всех элементов формы вызова функции, включая сам элемент-функцию. Так как ограничений нет, то компилятор волен выбирать любой устраивающий его порядок. [**см. стр. 202**] А пользователь, в свою очередь, не может рассчитывать на какой-либо определённый порядок вычислений и должен использовать `begin`, чтобы задать необходимый порядок явно.

Считается плохим стилем использовать вызовы функций, чтобы получить побочные эффекты в нужной последовательности. Поэтому следует избегать выражений вроде `(f (set! f` π`) (set! f` π'`))`, где неясно, какая же функция будет вызвана на самом деле. Ошибки, возникающие в подобных случаях, очень сложно отлавливать.

Окружение исполнения функций

Применение функции сводится к вычислению выражений, составляющих её тело, в окружении, где аргументы функции связаны со значениями, переданными при вызове функции. Вспомните, что при вызове `make-function` мы передали всё необходимое для этого, находящееся в распоряжении `evaluate`. В оставшейся части этого раздела мы будем разбирать используемые при вычислениях окружения, в программах они будут набраны *курсивом*.

[13] Функция могла бы потом ещё проверять на правильность типы переданных аргументов, но это не имеет отношения к механизму вызова.

Минимальное окружение

Для начала рассмотрим минимально возможное окружение:

```
(define (make-function variables body env)
  (lambda (values)
    (eprogn body (extend env.init variables values)) ) )
```

В строгом соответствии с ранее описанным соглашением, тело функции вычисляется в окружении, где аргументы функции связаны с переданными значениями. Например, получив в результате вызова `make-function` комбинатор `K`, определяемый как `(lambda (a b) a)`, мы можем его вызвать следующим образом:

```
(K 1 2) → 1
```

Но есть и неприятность: функция может использовать только свои аргументы и локальные переменные, потому что мы определили *env.init* как пустое окружение. В нём нет даже базовых функций из глобального окружения вроде `car` или `cons`.

Улучшенное окружение

Хорошо, попробуем улучшить наше определение следующим образом:

```
(define (make-function variables body env)
  (lambda (values)
    (eprogn body (extend env.global variables values)) ) )
```

Замечательно, теперь наши функции имеют доступ к глобальному окружению и всем его функциям. А что если мы попробуем определить взаимно рекурсивные функции? Также, какой результат даст программа слева (справа она же с раскрытыми макросами)?

```
(let ((a 1))                    ((lambda (a)
  (let ((b (+ 2 a)))              ((lambda (b)
    (list a b) ) )        ≡           (list a b) )
                                    (+ 2 a) ) )
                             1 )
```

Давайте рассмотрим по шагам, как вычисляется это выражение:

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)  | env.global
≡ ((lambda (b) (list a b)) (+ 2 a))                 | a → 1
                                                    | env.global
≡ (list a b)                                        | b → 3
                                                    | env.global
```

Тело внутренней функции `(lambda (b) (list a b))` выполняется в окружении, полученном расширением глобального окружения переменной `b`. Всё верно. Но в этом окружении нет необходимой переменной `a`!

Улучшенное окружение (вторая попытка)

Так как нам надо видеть переменную `a` во внутренней функции, то достаточно будет передать `invoke` текущее окружение, а она в свою очередь передаст его вызываемой функции. Чтобы реализовать эту идею, надо немного подправить `evaluate` и `invoke`; чтобы не путать эти определения с предыдущими, пусть они начинаются на `d.`:

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else     (d.invoke (d.evaluate (car e) env)
                            (evlis (cdr e) env)
                            env )) ) ) )

(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )

(define (d.make-function variables body def.env)
  (lambda (values current.env)
    (eprogn body (extend current.env variables values)) ) )
```

В этом определении стоит заметить, что передача окружения определения `env` через переменную *def.env* бессмысленна, так как при вызове используется лишь текущее окружение *current.env*.

Давайте теперь ещё раз рассмотрим пример, приведённый выше. Сейчас переменные не пропадают:

```
((lambda (a) ((lambda (b) (list a b)) (+ 2 a))) 1)|
                                                  | env.global
≡ ((lambda (b) (list a b)) (+ 2 a))|
                                   | a → 1
                                   | env.global
≡ (list a b)|
            | b → 3
            | a → 1
            | env.global
```

Заодно мы явно видим *стек вызовов*: каждая связывающая форма сначала укладывает свои новые переменные поверх текущего окружения, а потом убирает их оттуда после окончания вычислений.

Исправляем проблему

Но даже при таком определении всё ещё есть проблемы. Рассмотрим следующий пример:

```
(((lambda (a)
    (lambda (b) (list a b)) )
  1 )
 2 )
```

Функция `(lambda (b) (list a b))` создаётся в окружении, где `a` связана со значением `1`, но в момент вызова в окружении будет присутствовать только `b`. Таким образом, мы опять потеряли переменную `a`.

Без сомнения, вы заметили, что в определении `d.make-function` присутствуют два окружения: окружение определения *def.env* и окружение исполнения *current.env*. В жизни функции есть два важных события: её создание и её вызов(ы). Очевидно, что создаётся функция только однажды, а вызываться может несколько раз; или вообще никогда не вызываться. Следовательно, единственное[14] окружение, которое мы однозначно можем связать с функцией, — это окружение, в котором она была создана. Вернёмся к исходным определениям функций `evaluate` и `invoke`, но в этот раз функцию `make-function` запишем следующим образом:

```
(define (make-function variables body env)
  (lambda (values)
    (eprogn body (extend env variables values)) ) )
```

Теперь все приведённые примеры работают нормально. В частности, пример выше вычисляется следующим образом:

```
(((lambda (a) (lambda (b) (list a b))) 1) 2)  | env.global
≡ ((lambda (b) (list a b))                    | a → 1
                                              | env.global
    2 )                                       | env.global
≡ (list a b)                                  | b → 2
                                              | a → 1
                                              | env.global
```

Форма `(lambda (b) (list a b))` создаётся в глобальном окружении, расширенном переменной `a`. Когда эта функция вызывается, она расширяет окружение своего создания переменной `b`, таким образом, тело функции будет вычисляться в окружении, где обе переменные `a` и `b` присутствуют. После того, как функция вернёт результат, исполнение продолжается в глобальном

[14] На самом деле, здесь можно использовать любое необходимое окружение. См. про форму `closure` на странице 144.

окружении. Мы будем называть значение абстракции *замыканием* (closure), потому что при создании этого значения тело функции становится замкнутым в окружении своего определения.

Стоит отметить, что сейчас `make-function` сама использует замыкания языка определения. Это не является обязательным, как мы покажем далее в третьей главе. [**см. стр. 120**] Функция `make-function` возвращает замыкания, а это — характерная черта функциональных языков программирования.

1.6.1. Динамическая и лексическая области видимости

Из этого разговора об окружениях можно сделать два вывода. Во-первых, ясно, что с окружениями не всё так просто. Любое вычисление всегда производится в каком-то окружении, следовательно, необходимо эффективно реализовывать их использование. В третьей главе рассматриваются более сложные вещи вроде раскрутки стека и соответствующей формы `unwind-protect`, которые потребуют от нас ещё более точного контроля над окружениями.

Второй момент связан с двумя рассмотренными в предыдущем разделе вариантами, которые являются примерами *лексического* и *динамического связывания*[15] (также применяются термины лексическая и динамическая область видимости). В *лексическом* Лиспе функция выполняется в окружении своего определения, расширенном собственными переменными, тогда как в *динамическом* — расширяет текущее окружение, окружение своего вызова.

Сейчас в моде лексическое связывание, но это не значит, что у динамического нет будущего. С одной стороны, именно динамическое связывание применяется в некоторых довольно популярных языках вроде TEX [Knu84], Emacs Lisp* [LLSt93], Perl [WS91].

С другой стороны, сама идея динамической области видимости является важной концепцией программирования. Она соответствует установке связей перед выполнением вычислений и гарантированному автоматическому удалению этих связей после завершения вычислений.

Такую стратегию можно эффективно применять, например, в искусственном интеллекте. В этом случае сначала выдвигается некая гипотеза, затем из неё вырабатываются следствия. Как только система натыкается на противоречие, то гипотезу следует отвергнуть и перейти к следующей. Это называется *поиском с возвратом*. Если следствия гипотез хранятся без использования побочных эффектов, например, в A-списках, то отвержение гипотезы автоматически и без проблем утилизирует и все её следствия. Но если для этого используются глобальные переменные, массивы и т. д., то тогда за ненужной гипотезой приходится долго убирать, вспоминая, каким же было состояние

[15] В объектно-ориентированных языках под динамическим связыванием обычно понимается механизм выбора метода объекта на основе его реального типа во время исполнения программы, в противоположность статическому связыванию, при котором метод выбирается компилятором исходя из типа переменной, которая хранит рассматриваемый объект.

* Начиная с Emacs Lisp v. 24 и Perl 5, эти языки имеют и лексические переменные.

памяти в момент формулировки гипотезы и какие его части можно откатить до старых значений, чтобы ничего не сломать! Динамическая область видимости позволяет гарантировать существование переменной с определённым значением на время и только во время вычислений, независимо от того, будут они успешны или нет. Это свойство также широко используется при обработке исключений.

Область видимости переменной — это, можно сказать, географическое понятие в программе: местность, где переменная встречается и её можно использовать. В чистом Scheme (не обременённом полезными, но не абсолютно необходимыми вещами вроде `let`) есть только одна связывающая форма: `lambda`. Это единственная форма, вводящая новые переменные и предоставляющая им область видимости в рамках определяемой функции. В динамическом же Лиспе область видимости в принципе не может быть ограничена функцией. Рассмотрим следующий пример:

```
(define (foo x) (list x y))
(define (bar y) (foo 1991))
```

В лексическом Лиспе переменная `y` в `foo`[16] — это всегда ссылка на глобальную переменную `y`, которая не имеет никакого отношения к `y` внутри `bar`. В динамическом же Лиспе переменная `y` из `bar` будет видима в `foo` внутри `bar`, потому что в момент вызова `foo` переменная `y` уже находилась в текущем окружении. Следовательно, если мы дадим глобальной `y` значение `0`, то получим следующие результаты:

```
(define y 0)
(list (bar 100) (foo 3)) → ((1991 0) (3 0))    ; в динамическом Лиспе
(list (bar 100) (foo 3)) → ((1991 100) (3 0)) ; в лексическом Лиспе
```

Заметьте, что в динамическом Лиспе `bar` понятия не имеет о том, что в `foo` используется её же локальная переменная `y`, а `foo` не знает о том, в каком именно окружении следует искать значение своей свободной переменной `y`. Просто `bar` при вызове положила в текущее окружение переменную `y`, а внутренняя функция `foo` нашла её в своём текущем окружении. Непосредственно перед выходом `bar` уберёт свою `y` из окружения, и глобальная переменная `y` снова станет видна.

Конечно, если не использовать свободные переменные, то нет особой разницы между динамической и лексической областями видимости.

Лексическое связывание получило своё имя потому, что в данном случае достаточно иметь только код функции, чтобы с уверенностью отнести каждую используемую в ней переменную к одному из двух классов: или переменная находится внутри связывающей формы и является локальной, или же это глобальная переменная. Это чрезвычайно просто: достаточно взять исходный код, взять карандаш (или мышку) и поставить его кончик на переменную,

[16] О происхождении *foo* см. [Ray91].

значение которой нас интересует, после чего следует вести карандаш справа налево, снизу вверх до тех пор, пока не встретим первую связывающую форму. Динамическое же связывание названо в честь концепции *динамического времени жизни* переменных, которую мы будем рассматривать позже. [**см. стр. 103**]

Scheme поддерживает только лексические переменные. Common Lisp поддерживает оба типа с одинаковым синтаксисом. Синтаксис EuLisp и ISLisp разделяет эти два типа переменных, и они находятся в отдельных пространствах имён. [**см. стр. 64**]

Область видимости переменной может прерываться. Такое случается, когда одна переменная *скрывает* другую из-за того, что обе имеют одинаковое имя. Лексические области видимости вкладываются друг в друга, скрывая переменные с совпадающими именами из внешних областей. Этот известный «блокирующий» порядок разрешения конфликтов унаследован от Алгола 60.

Под влиянием λ-исчисления, в честь которого названа специальная форма `lambda` [Per79], Lisp 1.0 был сделан динамическим, но вскоре Джон Маккарти осознал, что он ожидал получить от следующего выражения `(2 3)`, а не `(1 3)`:

```
(let ((a 1))
  ((let ((a 2)) (lambda (b) (list a b)))
   3 ) )
```

Эта аномалия (не осмелюсь назвать её ошибкой) была исправлена введением новой специальной формы `function`. Она принимала `lambda`-форму и создавала *замыкание* — функцию, связанную с окружением, в котором она определена. При вызове замыкания вместо текущего окружения расширялось окружение определения, замкнутое внутри него. Вместе с изменениями `d.evaluate` и `d.invoke`, форма `function`[17] выражается так:

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((function)   ; Синтаксис: (function (lambda аргументы тело))
         (let* ((f   (cadr e))
                (fun (d.make-function (cadr f) (cddr f) env)) )
           (d.make-closure fun env) ) )
        ((lambda) (d.make-function (cadr e) (cddr e) env))
        (else     (d.invoke (d.evaluate (car e) env)
                            (evlis (cdr e) env)
                            env )) ) ) )
```

[17] Наша имитация не совсем точна, так как существует немало диалектов Лиспа (вроде CLtL1 [Ste84]), где `lambda` — это не специальный оператор, а только ключевое слово-маркер вроде `else` внутри `cond` и `case`. В этом случае `d.evaluate` может вообще не знать ни о какой `lambda`. Иногда даже накладываются ограничения на положение `lambda`-форм, разрешающие им находиться только внутри `function` и в определениях функций.

```
(define (d.invoke fn args env)
  (if (procedure? fn)
      (fn args env)
      (wrong "Not a function" fn) ) )

(define (d.make-function variables body env)
  (lambda (values current.env)
    (eprogn body (extend current.env variables values)) ) )

(define (d.make-closure fun env)
  (lambda (values current.env)
    (fun values env) ) )
```

Но это ещё не конец всей истории. `function` — это лишь костыль, на который опиралась хромая реализация Лиспа. С созданием первых компиляторов стало ясно, что с точки зрения производительности у лексической области видимости есть (ожидаемое при компиляции) преимущество: можно сгенерировать код для более-менее прямого доступа к любой переменной, а не динамически отыскивать её значение заново каждый раз. Тогда по умолчанию стали делать все переменные лексическими, за исключением тех, которые были явно помечены как динамические или, как тогда их называли, *специальные*. Выражение `(declare (special` x`))` являлось командой компиляторам Lisp 1.5, Common Lisp, Maclisp и других, говорившей, что переменная x ведёт себя «особенно».

Эффективность была не единственной причиной принятия такого решения. Другой причиной была потеря *ссылочной прозрачности* (referential transparency). Ссылочная прозрачность — это свойство языка, заключающееся в том, что замена в программе любого выражения его эквивалентом никак не изменит поведение этой программы (оба варианта программы или вернут одно и то же значение, или вместе застрянут в бесконечном цикле). Например:

`(let ((x (lambda () 1))) (x))` ≡ `((let ((x (lambda () 1))) x))` ≡ `1`

В общем случае ссылочная прозрачность теряется, если язык позволяет побочные эффекты. Чтобы она сохранилась и при наличии побочных эффектов, необходимо точнее определить понятие эквивалентных выражений. Scheme обладает ссылочной прозрачностью, если не использовать присваивания, функции с побочными эффектами и продолжения. [**см. упр. 3.10**] Это свойство желаемо и в наших программах, если мы хотим сделать их по-настоящему повторно используемыми, как можно менее зависимыми от контекста использования.

Локальные переменные функций вроде `(lambda (u) (+ u u))` иногда называются *безымянными*. Их имена ничего не значат и могут быть абсолютно произвольными. Функция `(lambda (n347) (+ n347 n347))` — это та же самая[18] функция, что и `(lambda (u) (+ u u))`.

[18] В терминах λ-исчисления подобная замена имён называется α-конверсией.

Мы ожидаем, что в языке будет сохраняться этот инвариант. Но это невозможно в динамическом Лиспе. Рассмотрим следующий пример:

```
(define (map fn l)  ; или mapcar, как кому нравится
  (if (pair? l)
      (cons (fn (car l)) (map fn (cdr l)))
      '() ) )

(let ((l '(a b c)))
  (map (lambda (x) (list-ref l x))
       '(2 1 0)))
```

(Функция `(list-ref` ℓ n`)` возвращает n-й элемент списка ℓ.)

В Scheme мы бы получили `(c b a)`, но в динамическом Лиспе результатом будет `(0 0 0)`! Причина: свободная переменная `l` в функции `(lambda (x) (list-ref l x))`, имя которой уже занято локальной переменной `l` в `map`.

Это затруднение можно решить, просто изменив конфликтующие имена. Например, достаточно будет переименовать какую-нибудь из двух `l`. Например, ту, которая внутри `map`, потому что это более разумно. Но какое имя выбрать, чтобы эта проблема не возникла снова? Если приписывать спереди к имени каждой переменной номер паспорта программиста, а сзади — текущее Un⋆x-время, то это, конечно, значительно снизит вероятность коллизий, но читабельность программ будет оставлять желать лучшего.

В начале восьмидесятых годов сложилась довольно неприятная ситуация: студентов учили Лиспу на примере интерпретаторов, но их понимание областей видимости отличалось от понимания компиляторов. В 1975 году Scheme [SS75] показал, что интерпретатор и компилятор возможно примирить, поместив обоих в мир, где все переменные лексические. Common Lisp забил последний гвоздь в гроб этой проблемы, постановив, что *хорошее* понимание — это понимание компилятора, а для него удобнее лексические переменные. Интерпретатор должен был подчиниться новым правилам. Растущий успех Scheme и других функциональных языков, вроде ML и компании, популяризовал новый подход сначала в языках программирования, а затем и в умах людей.

1.6.2. Дальнее и ближнее связывание

Но не всё так просто заканчивается. Разработчики языков нашли способы ускорить поиск значений динамических переменных. Если окружения представлены ассоциативными списками, то время на поиск значения переменной (стоимость вызова `lookup`) линейно зависит от длины списка.[19] Такой подход

[19] К счастью, статистика показывает, что переменные, располагающиеся ближе к началу списка, используются чаще тех, что находятся глубоко внутри. Кстати, ещё стоит отметить, что лексические окружения в среднем меньше по размеру, чем динамические, так как последним необходимо хранить все переменные, участвующие в вычислениях, включая одноимённые [Bak92a].

называется *глубоким* или *дальним связыванием* (deep binding), так как значения динамических переменных обычно располагаются на некотором удалении от текущего локального окружения.

Существует и другой метод, называемый *поверхностным* или *ближним связыванием* (shallow binding). Суть его в том, что переменная напрямую связана с местом, где хранится её значение в текущий момент, без привязки к окружению. Проще всего это реализовать, положив это значение в специальное поле символа, соответствующего этой переменной; это поле называют `Cval` или *ячейкой значения* (value cell). В таком случае стоимость `lookup` постоянна или около того: требуется лишь одна косвенная адресация и, может быть, сдвиг. Так как бесплатный сыр бывает только в мышеловке, то стоит отметить, что вызов функции при использовании этого метода выходит дороже, потому что требуется сначала где-то сохранить старые значения аргументов, затем записать новые значения в поля соответствующих символов. А потом, что самое важное, после выхода из функции старые значения в символах необходимо восстановить обратно, а это может помешать оптимизации хвостовой рекурсии. (Хотя есть варианты: [SJ93].)

Изменив структуру окружений, мы сможем частично проэмулировать[20] ближнее связывание. Но с оговорками: список аргументов не может быть точечным (так будет легче его разбирать) и мы не будем проверять арность функций. Новые функции будем обозначать префиксом `s.`, чтобы не путать их с другими.

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (let ((old-bindings
           (map (lambda (var val)
                  (let ((old-value (getprop var 'apval)))
                    (putprop var 'apval val)
                    (cons var old-value) ) )
                variables
                values ) ))
      (let ((result (eprogn body current.env)))
        (for-each (lambda (b) (putprop (car b) 'apval (cdr b)))
                  old-bindings )
        result ) ) ) )

(define (s.lookup id env)
  (getprop id 'apval) )

(define (s.update! id env value)
  (putprop id 'apval value) )
```

[20] Здесь мы не реализуем присваивание переменным, захваченным замыканиями. Об этом можно почитать в [BCSJ86].

В Scheme функции `putprop` и `getprop` не входят в стандарт, так как здесь не любят неэффективные глобальные побочные эффекты, но тем не менее, даже в [AS85] есть аналогичные `put` и `get`. [**см. упр. 2.6**]

С помощью этих функций мы эмулируем наличие у символов поля,[21] где хранится значение одноимённой переменной. Независимо от их настоящей реализации,[22] будем считать, что они выполняются за постоянное время.

Заметьте, что в этой реализации абсолютно не используется окружение определения `env`. Поэтому для поддержки замыканий нам потребуется изменить реализацию `make-closure`, так как она теперь не имеет доступа к окружению определения (ввиду его отсутствия). При создании замыкания необходимо просмотреть тело функции, выделить все свободные переменные и правильно их сохранить внутри замыкания. Мы реализуем это позже.

Дальнее связывание облегчает смену окружений и многозадачность, теряя в скорости поиска переменных. Ближнее связывание ускоряет поиск переменных, но теряет в скорости вызова функций. Генри Бейкеру [Bak78] удалось объединить эти два подхода в технику под названием *rerooting*.

Наконец, не забывайте, что ближнее и дальнее связывание — это лишь способы реализации, они никак не влияют на само понятие связывания.

1.7. Глобальное окружение

Пустое глобальное окружение — это печально, поэтому большинство лисп-систем предоставляют *библиотеки* функций. Например, в глобальном окружении Common Lisp (CLtL1) около 700 функций, у Le_Lisp их более 1500, у ZetaLisp — более 10 000. Без библиотек Лисп был бы лишь прикладным λ-исчислением, в котором нельзя даже распечатать полученные результаты. Библиотеки очень важны для конечного пользователя. Специальные формы — это строительные кирпичики для разработчиков интерпретаторов, но для конечного пользователя такими кирпичиками являются функции библиотек. По-видимому, именно отсутствие в чистом Лиспе таких банальных вещей вроде библиотеки тригонометрических функций прочно укоренило мысль о непригодности Лиспа для «серьёзных программ». Как говорится в [Sla61], возможность символьного интегрирования или дифференцирования — это, конечно, замечательно, но кому нужен язык, где нет даже синуса или тангенса?

Мы ожидаем, что все привычные функции вроде `cons`, `car` и т. п. будут доступны в глобальном окружении. Также можно туда поместить несколько простых констант вроде логических значений и пустого списка.

[21] Это поле названо в честь `apval` из [MAE+62]. [**см. стр. 51**] Тогда значения полей действительно хранились в наивных P-списках.

[22] Эти функции проходят по списку свойств символа (его P-списку, от property) до тех пор, пока не найдут нужное. Скорость поиска, соответственно, линейно зависит от длины списка, если только не применяются хеш-таблицы.

Для этого мы определим пару макросов. Исключительно для удобства, потому что мы о них ещё даже не говорили.[23] Макросы — это довольно сложная и важная вещь сами по себе, так что им посвящена собственная глава. [**см. стр. 370**]

Эти два макроса облегчат наполнение глобального окружения. Само глобальное окружение является расширением начального окружения `env.init`.

```
(define env.global env.init)

(define-syntax definitial
  (syntax-rules ()
    ((definitial name)
     (begin (set! env.global (cons (cons 'name 'void) env.global))
            'name ) )
    ((definitial name value)
     (begin (set! env.global (cons (cons 'name value) env.global))
            'name ) ) ) )

(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial name
        (lambda (values)
          (if (= arity (length values))
              (apply value values)      ; Родная apply Scheme
              (wrong "Incorrect arity" (list 'name values) ) ) ) ) ) ) )
```

Несмотря на то, что стандарт Scheme этого не требует, мы определим несколько полезных констант. Заметим, что `t` — это переменная в определяемом Лиспе, а `#t` — это значение из определяющего Лиспа. Оно подходит, так как любое значение, не совпадающее с `the-false-value`, является *истиной*.

```
(definitial t #t)
(definitial f the-false-value)
(definitial nil '())
```

Хотя это удобно — иметь глобальные переменные с настоящими объектами для данных сущностей, но есть и другое решение: особый синтаксис. Scheme использует `#t` и `#f`, подставляя вместо них логические *истину* и *ложь*. В этом есть определённый смысл:

1) Они всегда видимы: `#t` означает *истину* в любом контексте, даже тогда, когда локальная переменная названа `t`.

2) Значение `#t` невозможно изменить, но многие интерпретаторы позволят изменить значение глобальной переменной `t`.

Например, выражение `(if t 1 2)` вернёт 2, если оно вычисляется в следующем окружении: `(let ((t #f)) (if t 1 2))`.

[23] Согласитесь, было бы странным втискивать всю книгу в первую главу.

Существует много способов ввести такой синтаксис. Наиболее простой способ — это вшить значения `t` и `f` в вычислитель:

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((eq? e 't) #t)
            ((eq? e 'f) #f)
            ...
            ((symbol? e) (lookup e env))
            ...
            (else (wrong "Cannot evaluate" exp)) )
      ... ) )
```

Также мы могли бы ввести понятия *изменяемого* и *неизменяемого* связывания. Неизменяемые переменные отвергаются присваиванием. Ничто и никогда не сможет изменить значение неизменяемой переменной. Такая концепция существует, хоть и не всегда явно, во многих системах. Например, существуют так называемые *инлайн-функции* (также известные как *подставляемые* или *встраиваемые*), вызов которых можно полностью заменить прямой подстановкой их тела. [**см. стр. 240**]

Чтобы можно было спокойно подставить вместо `(car x)` код функции, возвращающей левый элемент точечной пары `x`, необходимо быть абсолютно уверенным в том, что значение глобальной переменной `car` никогда не менялось и не поменяется в будущем. Посмотрите, какая беда случается, если это не так:

```
(set! my-global (cons 'c 'd))
   → (c . d)
(set! my-test (lambda () (car my-global)))
   → #<MY-TEST procedure>
(begin (set! car cdr)
       (set! my-global (cons 'a 'b))
       (my-test) )
   → ?????
```

К счастью, в результате может получиться только `a` или `b`. Если `my-test` использует значение `car` на момент определения, то мы получим `a`. Если же `my-test` будет использовать текущее значение `car`, то ответом будет `b`. Полезным будет также сравнить в этом аспекте `my-test` и `my-global`: обычно первый вариант поведения ожидается от `my-test` при использовании компилятора, тогда как для `my-global` нормальным считается именно второй вариант. [**см. стр. 77**]

Также мы добавим несколько рабочих переменных[24] в глобальное окружение, так как сейчас у нас нет способа динамически создавать переменные. По

[24] К сожалению, сейчас они ещё и инициализируются. Эта ошибка будет исправлена позже.

статистике, предлагаемые имена составляют приблизительно 96,037 % используемых при тестировании свеженаписанных интерпретаторов.

```
(definitial foo)
(definitial bar)
(definitial fib)
(definitial fact)
```

Наконец, определим несколько примитивных функций (не все, потому что такие полные списки — это хорошее снотворное). Главная сложность состоит в соединении механизмов вызова функций определяемого и определяющего языков. Зная, что аргументы собираются нашим интерпретатором в список, достаточно просто применить к нему `apply`.[25] Заметьте, что арность функций будет соблюдаться, так как мы включили проверку в определение макроса `defprimitive`.

```
(defprimitive cons cons 2)
(defprimitive car car 1)
(defprimitive set-cdr! set-cdr! 2)
(defprimitive + + 2)
(defprimitive eq? eq? 2)
(defprimitive < < 2)
```

1.8. Запускаем интерпретатор

Нам осталось показать только одну вещь: дверь в наш новый мир.

```
(define (chapter1-scheme)
  (define (toplevel)
    (display (evaluate (read) env.global))
    (toplevel) )
  (toplevel) )
```

Поскольку наш интерпретатор ещё мал и неопытен, но подаёт большие надежды, предлагаем вам в качестве упражнения написать функцию, позволяющую из него выйти.

1.9. Заключение

Действительно ли мы сейчас определили язык?

Нет никаких сомнений в том, что мы можем запустить `evaluate`, передать ей выражение, и она вскоре вернёт результат вычислений. Но сама функция `evaluate` не имеет никакого смысла без языка своего определения, а если у нас нет определения языка определения, то мы вообще ни в чём не можем быть

[25] Можно только порадоваться за наш выбор не называть `invoke` «`apply`».

уверены. Так как каждый лиспер является дальним родственником барона Мюнхгаузена, то, наверное, будет достаточно взять в качестве языка определения тот, который мы только что определили. Следовательно, у нас есть язык L, определённый функцией `evaluate`, написанной на языке L. Такой язык является решением следующего уравнения относительно L:

$$\forall \pi \in \textbf{Программы}: L\texttt{(evaluate (quote } \pi\texttt{) env.global)} \equiv L\pi$$

Исполнение любой программы π, написанной на L (обозначается как $L\pi$), должно вести себя так же (то есть давать тот же результат или никогда не завершаться), как и выражение `(evaluate (quote` π`) env.global)` на том же языке L. Одним из занимательных следствий этого утверждения является то, что `evaluate` способна[26] проинтерпретировать сама себя. Следовательно, следующие выражения эквивалентны:

```
(evaluate (quote π) env.global) ≡
  ≡ (evaluate (quote (evaluate (quote π) env.global)) env.global)
```

Есть ли ещё решения приведённого уравнения? Да, и их великое множество! Как мы видели раньше, определение `evaluate` вовсе не обязательно указывает порядок вычислений. Множество других свойств языка, используемого для определения, бессознательно *наследуются* определяемым языком. Мы, по сути, ничего не можем о них сказать, но все эти варианты претендуют на решение указанного уравнения. Вместе с многочисленными тривиальными решениями. Рассмотрим, к примеру, язык L_{2001}, любая программа на котором возвращает 2001. Даже такой язык удовлетворяет этому уравнению. Поэтому для определения настоящих языков необходимы другие методы, их мы рассмотрим в следующих главах.

1.10. Упражнения

Упражнение 1.1 Модифицируйте функцию `evaluate` так, чтобы она стала трассировщиком. Все вызовы функций должны выводить на экран фактические аргументы и возвращаемый результат. Легко представить себе дальнейшее развитие этого инструмента в пошаговый отладчик, вдобавок позволяющий изменять порядок выполнения отлаживаемой программы.

Упражнение 1.2 Если функции `evlis` передаётся список из одного выражения, она делает один лишний рекурсивный вызов. Придумайте способ, как избавиться от него.

Упражнение 1.3 Предположим, новая функция `extend` определена так:

[26] После того, как мы раскроем все используемые макросы и сокращения вроде `let`, `case`, `define` и т. д. Потом надо будет ещё поместить в глобальное окружение функции `evaluate`, `evlis` и др.

```
(define (extend env names values)
  (cons (cons names values) env) )
```

Определите соответствующие функции `lookup` и `update!`. Сравните их с ранее рассмотренными.

Упражнение 1.4 В работе [SS80] предлагается другой механизм ближнего связывания, названный *rack*. Символ связывается с полем, хранящим не единственное значение, а стек значений. В каждый момент времени значением переменной является находящаяся на вершине стека величина. Перепишите функции `s.make-function`, `s.lookup` и `s.update!` для реализации этой идеи.

Упражнение 1.5 Если вы ещё не заметили, то в определение функции `<` вкралась ошибка! Ведь эта функция должна возвращать логические значения определяемого языка, а не определяющего. Исправьте это досадное недоразумение.

Упражнение 1.6 Определите функцию `list`.

Упражнение 1.7 Для обожающих продолжения: определите `call/cc`.

Упражнение 1.8 Определите функцию `apply`.

Упражнение 1.9 Определите функцию `end`, позволяющую выйти из интерпретатора, разработанного в этой главе.

Упражнение 1.10 Сравните скорость Scheme и `evaluate`. Затем сравните скорость `evaluate` и `evaluate`, интерпретируемой с помощью `evaluate`.

Упражнение 1.11 Ранее мы смогли успешно определить `begin` через `lambda` [**см. стр. 29**], но для этого нам потребовалось использовать функцию `gensym`, чтобы избежать коллизий имён переменных. Переопределите `begin` в таком же духе, но без использования `gensym`.

Рекомендуемая литература

Все работы по интерпретаторам, приведённые в начале этой главы, являются довольно интересными, но если вы не можете столько читать, то вот наиболее стоящие из них:

- среди «λ-papers»: [SS78a];
- самая короткая в мире статья, которая содержит полный интерпретатор Лиспа: [McC78b];
- «нестрого формальное» описание интерпретации: [Rey72];
- местная книга Бытия: [MAE+62].

Глава 2

Lisp, 1, 2, . . . , ω

Функции занимают центральное место в Лиспе, поэтому очевидно, что эффективность их вызовов очень важна. Неудивительно, что за прошедшее время соответствующие механизмы были изучены вдоль и поперёк, а исследования по диагонали всё ещё продолжаются. В этой главе речь пойдёт о различных вариантах понимания функций и их вызовов. Мы поговорим о том, что называется Lisp_1 и Lisp_2, а также их различиях, вызываемых концепцией отдельных пространств имён. Заканчивается глава рассмотрением рекурсии и способов её реализации с учётом изученных вопросов.

Среди всевозможных объектов, используемых интерпретатором, функции требуют особого подхода. Этот базовый тип объектов имеет личный конструктор: `lambda`, и поддерживает как минимум одну операцию: применение функции к аргументам. Вряд ли можно его описать ещё проще, не сделав совсем бесполезным. Кстати, именно то, что у функций немного характерных черт, делает их прекрасным строительным блоком, инкапсулирующим поведение; функция может делать только то, для чего она запрограммирована. К примеру, с помощью функций можно представлять объекты, имеющие поля и методы [AR88]. В Scheme функциями представляется вообще всё, что только можно представить в виде функции.

Попытки сделать вызовы функции более эффективными привели к множеству (часто несовместимых) вариаций языка. Изначально Lisp 1.5 [MAE+62] не имел понятия объекта-функции. Реализация была такова, что переменная, функция и макрос могли одновременно носить одно и то же имя, так как хранились в различных ячейках (`APVAL`, `EXPR` и `MACRO`[1]) списка свойств соответствующего символа.

Maclisp выделял именованные функции в отдельную категорию, а его потомок Common Lisp (CLtL2) [Ste90] только недавно получил поддержку функций как объектов первого класса. В Common Lisp `lambda` — это ключевое слово со значением: «Внимание! Дальше идёт определение анонимной функции». `lambda`-формы не имеют возвращаемого значения и могут находиться

[1] `APVAL`, *A Permanent VALue*, для глобальных переменных; `EXPR`, *EXPRession*, для глобальных функций; `MACRO` для макросов.

лишь в определённых местах: как первый элемент формы вызова или как первый аргумент специальной формы `function`.

В отличие от них, Scheme с самой первой версии 1975 года имел функциональные объекты и единое пространство имён, распространяя понятие *первого класса* практически на всё. Объекты первого класса (полноценные объекты) могут приниматься или возвращаться функциями, находиться в списке, массиве, переменной и т. д. Такой подход широко распространён в классе языков вроде ML, он же будет использоваться и здесь.

2.1. Lisp_1

В предыдущей главе мы применяли именно такой подход: функции были объектами (их создавала `make-function`); в процессе вычислений мы не различали функции и аргументы, при вычислении формы аппликации не было никакой разницы между вычислением элемента на месте *функции* и вычислением элементов, находящихся на месте *параметров*. Давайте ещё раз посмотрим на интерпретатор из предыдущей главы:

```
(define (evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((lambda) (make-function (cadr e) (cddr e) env))
        (else (invoke (evaluate (car e) env)
                      (evlis (cdr e) env) )) ) ) )
```

Самое важное в нём:

1) `lambda` создаёт полноценные объекты — замыкания, — которые сохраняют окружение своего определения.

2) При применении функции всё вычисляется одинаковым образом: с помощью `evaluate`; `evlis` лишь вызывает `evaluate` для всех элементов списка.

Благодаря этим двум качествам Scheme является представителем семейства Lisp_1.

2.2. Lisp_2

В большинстве программ на Лиспе при вызовах функций первым элементом формы является имя глобальной функции. Это справедливо и для программ из предыдущей главы. Мы могли бы сделать данное ограничение частью языка. Это бы не сильно изменило внешний вид кода, но облегчило бы вычисление форм: для первого элемента уже не нужна вся мощь `evaluate`,

достаточно мини-вычислителя, который бы умел только искать нужную функцию по имени. Для реализации этой идеи изменим соответствующую часть `evaluate`:

```
...
(else (invoke (lookup (car e) env)
              (evlis (cdr e) env) )) ...
```

Теперь у нас два разных вычислителя, по одному на каждый случай, где мы можем встретить переменную: на месте функции или на месте аргумента. Мы можем по-разному обрабатывать один и тот же идентификатор в зависимости от его положения. Если функции требуют особого подхода, то логичным будет также выделить для них особое пространство имён. Очевидно, что легче искать имена функций в окружении, где нет имён переменных, которые только мешают. Интерпретатор легко адаптировать для этого случая. Нам понадобится окружение для функций `fenv` и специализированный вычислитель `evaluate-application`, который знает, как обращаться с элементами данного окружения. Так как теперь у нас два окружения и два вычислителя, то мы назовём это Lisp_2 [SG93].

```
(define (f.evaluate e env fenv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (f.evaluate (cadr e) env fenv)
                      (f.evaluate (caddr e) env fenv)
                      (f.evaluate (cadddr e) env fenv) ))
        ((begin)  (f.eprogn (cdr e) env fenv))
        ((set!)   (update! (cadr e)
                           env
                           (f.evaluate (caddr e) env fenv) ))
        ((lambda) (f.make-function (cadr e) (cddr e) env fenv))
        (else     (evaluate-application (car e)
                                        (f.evlis (cdr e) env fenv)
                                        env
                                        fenv )) ) ) )
```

За вычисление форм отвечает `evaluate-application`, которая принимает «сырое» имя функции, вычисленные значения аргументов и два текущих окружения. Как видно из определения `lambda`, при создании функции замыкаются оба окружения: `env` и `fenv`. В остальном новая версия отличается только тем, что за `env` следует (как тень) `fenv`. Естественно, ещё необходимо доработать функции `evlis` и `eprogn`, чтобы они использовали `fenv`:

```
(define (f.evlis exps env fenv)
  (if (pair? exps)
      (cons (f.evaluate (car exps) env fenv)
            (f.evlis (cdr exps) env fenv) )
      '() ) )

(define (f.eprogn exps env fenv)
  (if (pair? exps)
    (if (pair? (cdr exps))
        (begin (f.evaluate (car exps) env fenv)
               (f.eprogn (cdr exps) env fenv) )
        (f.evaluate (car exps) env fenv) )
    empty-begin ) )
```

При вызове функции её аргументы расширяют окружение переменных, для этого необходимо доработать только способ создания функции; механизм вызова (`invoke`) изменений не требует.

```
(define (f.make-function variables body env fenv)
  (lambda (values)
    (f.eprogn body (extend env variables values) fenv) ) )
```

Задача `evaluate-application` в том, чтобы проанализировать функциональный элемент формы и обеспечить правильный вызов. Если мы последуем путём Common Lisp, то на месте функционального элемента может стоять или символ, или `lambda`-форма.

```
(define (evaluate-application fn args env fenv)
  (cond ((symbol? fn)
         (invoke (lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
        (else (wrong "Incorrect functional term" fn)) ) )
```

И что же мы в итоге получили, а что потеряли? Первое очевидное преимущество состоит в том, что для поиска функции по имени необходим только простой вызов `lookup`, а не `f.evaluate` с последующим долгим синтаксическим разбором. Далее, так как мы избавились ото всех ссылок на переменные в `fenv`, это окружение стало компактнее, а значит и поиск в нём ускорился. Второе преимущество состоит в ускорении вычисления форм, где на месте функции находится `lambda`-форма. Например:

```
(let ((state-tax 1.186))
  ((lambda (x) (* state-tax x)) (read)) )
```

В этом случае для `(lambda (x) (* state-tax x))` не будет создаваться замыкание, её тело будет вычислено сразу в правильном окружении.

Проблема в том, что эти два преимущества по сути ничего не дают, так как тех же результатов можно добиться и в Lisp_1 с помощью небольшого предварительного анализа. Есть только один действительно приятный момент: Lisp_2 чуть-чуть быстрее, так как мы можем быть уверены в том, что любое имя из `fenv` связано с функцией и ни с чем иным, а значит, проверку на то, что это действительно функция, надо выполнять лишь один раз: при помещении функции в окружение. И так как каждое имя должно быть связано с функцией, то все неиспользуемые имена можно просто связать с `wrong`.

Ввиду того, что каждое имя из `fenv` связано с функцией, мы можем вообще избавиться от вызова `invoke`, а заодно и от вызова `procedure?` внутри. Это возможно, потому что мы реализуем интерпретатор на Scheme (в Common Lisp формы вроде `((lookup fn fenv) args)` запрещены). Для этого следует немного изменить начало `evaluate-application`:

```
(define (evaluate-application fn args env fenv)
  (cond ((symbol? fn) ((lookup fn fenv) args))
        ... ) )
```

В Лиспе функции вызываются так часто, что любой выигрыш времени при вызовах — это уже хорошо и может сильно повлиять на производительность. Но этот конкретный выигрыш не так уж и велик: он появляется только для динамически определяемых функций, тогда как в большинстве случаев вызываемая функция известна статически.

Теперь поговорим о том, что же мы потеряли. А потеряли мы возможность *вычислить* применяемую функцию. Рассмотрим выражение `(if` *условие* `(+ 3 4) (* 3 4))`. В Scheme можно легко вынести аргументы `3` и `4` за скобки: `((if` *условие* `+ *) 3 4)`. Это просто, понятно и логично. Фактически, это тождество. Но в Lisp_2 данная программа вызовет ошибку, так как `if`-форма, стоящая на месте функции, — это ни символ, ни `lambda`-форма.

2.2.1. Вычисляем функции

Возможности текущего окружения функций ещё далеки от возможностей окружения переменных (окружения параметров). В частности, как вы видели, мы не можем вычислить применяемую функцию. Традиционный трюк, существующий ещё со времён Maclisp, заключается в том, что если `evaluate-application` натыкается на что-то непонятное, то она передаёт его функции `f.evaluate`:

```
(define (evaluate-application2 fn args env fenv)
  (cond ((symbol? fn)
         ((lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
```

```
        (else (evaluate-application2
               (f.evaluate fn env fenv) args env fenv)) ) )
```

Теперь наша проблема решается следующим образом:

`(if` *условие* `(+ 3 4) (* 3 4))` ≡ `((if` *условие* `'+ '*) 3 4)`

Не особо элегантно, так как необходимо писать лишние кавычки, но по крайней мере это работает. Работает, но, пожалуй, чересчур прилежно; иногда настолько, что попадает в бесконечный цикл.

`(''''''''''''''1789` *аргументы*`)`

Выражение `''''''''''''''1789` сначала будет вычислено тринадцать раз подряд, пока `evaluate-application2` не доберётся до числа 1789, а затем она попадёт в бесконечный цикл, так как 1789 — это не функция и не символ, а значит, его надо передать в `f.evaluate`, чтобы… получить опять число 1789! Короче говоря, надо тщательнее следить за тем, что передаётся `f.evaluate`. Проблема остаётся даже тогда, когда мы перепишем интерпретатор следующим образом:

```
(define (evaluate-application3 fn args env fenv)
  (cond
    ((symbol? fn)
     (let ((fun (lookup fn fenv)))
       (if fun (fun args)
           (evaluate-application3 (lookup fn env) args env fenv) ) ) )
    ... ) )
```

В этом случае, если символ отсутствует в функциональном окружении, то поиск повторяется уже в окружении переменных. Ой! Даже если предположить, что не существует функции `foo`, мы спокойно можем зациклиться при вычислении значения переменной. Например:

```
(let ((foo 'foo))
  (foo аргументы) )
```

Нет, можно, конечно, встроить в `evaluate-application` защиту от этого, проверяя, совпадает ли значение переменной с её именем. Но ведь можно написать и так:

```
(let ((flip 'flop)
      (flop 'flip) )
  (flip) )
```

Единственный вариант, который пока не удалось одурачить — это самый первый на странице 54, поэтому необходимо найти другой метод вычисления функционального элемента формы.

2.2.2. Двойственность миров

Обобщая данные затруднения, можно сказать, что есть некоторые вычисления, принадлежащие миру параметров, которые мы хотим перенести в мир

функций, и наоборот. Точнее говоря, мы хотим иметь возможность возвращать функцию как результат вычислений и передавать её как аргумент вычислений.

Если абсолютно необходимо, чтобы на месте функции стояло имя функции, но в то же время мы хотим иметь возможность вычислять необходимую функцию, то будет достаточно предопределённой функции, которая знает, как применять функции к аргументам. Назовём её `funcall` (от *function call*). Она применяет свой первый аргумент (который обязан быть функцией) к остальным. С её помощью наша проблема решается следующим образом:

`(if` *условие* `(+ 3 4) (* 3 4))` ≡ `(funcall (if` *условие* `+ *) 3 4)`

Все аргументы (первый в том числе) вычисляются обычной `f.evaluate`. Функция `funcall` только берёт всё готовое и выполняет вызов. Мы легко можем определить `funcall` как

```
(define (funcall . args)
  (if (> (length args) 1)
      (invoke (car args) (cdr args))
      (wrong "Incorrect arity" 'funcall) ) )
```

В Lisp_2 `funcall` используется для динамического вычисления функций. Во всех остальных случаях функция известна и не требуется проверять, действительно ли это функция.

Обратите внимание на вызов `invoke` в `funcall`. Эта функция проверяет, действительно ли её аргумент является функцией, в отличие от `evaluate-application`, где эта проверка не выполняется. Функций `funcall` больше похожа на `apply`: обе принимают первым аргументом функцию, а за ним — необходимые аргументы. Разница между ними лишь в том, что `funcall` статически знает количество передаваемых аргументов.

К сожалению, есть ещё одна небольшая проблема. Когда мы пишем `(if` *условие* `+ *)`, мы ожидаем получить функцию сложения или умножения. Но то, что мы сейчас получаем, — это значение переменной `+` или `*`! В Common Lisp значения этих переменных и близко не связаны с арифметикой: они хранят последнее считанное и последнее возвращённое REPL* значение!

Мы ввели `funcall` для того, чтобы получить возможность проводить вычисления с функциями как со значениями. Конечно же, в интерпретаторе присутствует и обратный процесс: ведь `evaluate-application` получает именно такие значения и вызывает их как функции; но эти преобразования происходят неявно. В присутствии `funcall` нам необходимо иметь возможность явно превратить имя функциональной переменной `+` в соответствующую функцию из окружения функций. Для этого мы введём ещё одно приспособление: `function`. Эта специальная форма принимает имя функции и возвращает со-

*REPL — «*read–eval–print loop*», интерактивная сессия. В интерпретаторе из предыдущей главы ей соответствует функция `toplevel`.

ответствующее функциональное значение. Теперь мы можем свободно перемещаться меж двух миров:

`(if` *условие* `(+ 3 4) (* 3 4)) ≡`
`≡ (funcall (if` *условие* `(function +) (function *)) 3 4)`

Для определения `function` достаточно добавить соответствующую обработку в `f.evaluate`. Эта форма `function` не имеет ничего общего с одноимённой [**см. стр. 41**] формой `(function (lambda` *переменные тело*`))`, создающей замыкания. Здесь мы определяем `(function` *имя-функции*`)`, преобразующую имена функций в их значения.

```
...
((function)
 (cond ((symbol? (cadr e))
        (lookup (cadr e) fenv) )
       (else (wrong "Incorrect function" (cadr e))) ) ) ...
```

Мы легко можем расширить это определение, чтобы оно принимало и `lambda`-формы, как это сделано в Common Lisp или у нас на странице 41. Но в этом нет особого смысла, так как наша `lambda` сразу возвращает нужное значение. В Common Lisp такое расширение необходимо, потому что там `lambda` лишь помечает код функции и «специальной форме» `lambda` позволено или находиться на месте функции, или быть первым аргументом специальной формы `function`.

Функция `funcall` позволяет поместить результат вычислений из мира параметров в переменную из мира функций. И наоборот, `function` позволяет поместить значение из мира функций в переменную из мира параметров. В них усматриваются очевидные параллели: «применение функций — `funcall`» и «разыменование переменных — `function`». Одновременное существование двух миров и возможность общения между ними требуют подобных мостов.

Заметьте, что сейчас мы не имеем возможности изменять функциональное окружение; у нас нет соответствующего присваивания. Это позволяет компиляторам без опаски и в полную силу задействовать механизм *инлайн-функций*. Одной из привлекательных сторон использования нескольких пространств имён является возможность наделить каждое из них особыми свойствами.

2.2.3. Используем Lisp_2

Чтобы полностью определить наш Lisp_2, остаётся указать начальное функциональное окружение и запустить в цикле сам интерпретатор, `f.evaluate`. Определение глобального функционального окружения мало чем отличается от окружения переменных: надо всего лишь расширять другое начальное окружение.

```
(define fenv.global '())
```

```
(define-syntax definitial-function
  (syntax-rules ()
    ((definitial-function name)
     (begin (set! fenv.global (cons (cons 'name 'void) fenv.global))
            'name) )
    ((definitial-function name value)
     (begin (set! fenv.global (cons (cons 'name value) fenv.global))
            'name) ) ) )

(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial-function name
       (lambda (values)
         (if (= arity (length values))
             (apply value values)
             (wrong "Incorrect arity"
              (list 'name 'values) ) ) ) ) ) ) )

(defprimitive car car 1)
(defprimitive cons cons 2)
```

И наконец:

```
(define ( бесспорно Lisp2 )
  (define (toplevel)
    (display (f.evaluate (read) env.global fenv.global))
    (toplevel) )
  (toplevel) )
```

2.2.4. Расширяем функциональное окружение

Все окружения — это объекты некоторого абстрактного типа данных. Что мы ожидаем от этого типа данных? Мы ожидаем, что он будет хранить связи между именами и сущностями, что мы можем по имени отыскать нужную сущность, и что мы можем добавлять новые связи. Ещё мы хотим иметь возможность определять локальные функции, а для этого необходим механизм локального расширения функционального окружения. В общем, хочется `let`, но только для функций. Сейчас функциональное окружение неизменяемо, так что было бы здорово иметь такую возможность. Создадим новую специальную форму `flet` (*functional let*) со следующим синтаксисом:

```
(flet ((имя1 аргументы1 тело1)
       ...
       (имяn аргументыn телоn) )
  выражения... )
```

Так как `flet` умеет создавать только локальные функции, нет необходимости писать `lambda`, это и так подразумевается. Специальная форма `flet` вычисляет формы (`lambda` *аргументы*$_i$ *тело*$_i$) и связывает получаемые значения с именами *имя*$_i$ в функциональном окружении. После этого *выражения* вычисляются в расширенном окружении, где можно ставить *имя*$_i$ на место функции или передавать его в `function`, если понадобится соответствующее замыкание.

Добавить `flet` в `f.evaluate` просто:

```
...
((flet)
 (f.eprogn
  (cddr e)
  env
  (extend fenv
          (map car (cadr e))
          (map (lambda (def)
                 (f.make-function (cadr def) (cddr def) env fenv) )
               (cadr e) ) ) ) ) ...
```

Форма `flet` серьёзно расширяет наши возможности: например, она позволяет `lambda` замыкать в себе не только `env`, но и `fenv`. Рассмотрим пример:

```
(flet ((square (x) (* x x))
  (lambda (x) (square (square x))) )
```

Значением этого выражения является анонимная функция, возводящая число в четвёртую степень. Это замыкание сохраняет в себе и использует локальную функцию `square`.

2.3. Другие возможности

Как только мы заставили вычислитель относиться по-особенному к функциям, на ум сразу же приходят другие варианты использования этой возможности. Например, мы могли бы трактовать числа как функции-аксессоры для списков:

```
(2 '(foo bar hux wok))  → hux
(-2 '(foo bar hux wok)) → (hux wok)
```

Число n считается синонимом для `cad`^{n}`r`, если оно неотрицательно, или `cd`$^{-n}$`r`, если оно меньше ноля. Базовым аксессорам `car` и `cdr` соответствуют числа `0` и `-1`. После этого элементарно реализуются чисто алгебраические преобразования `(-1 (-2 `π`))` $\equiv$ `(-3 `π`)` или `(2 (-3 `π`))` $\equiv$ `(5 `π`)`.

Также можно придать смысл спискам функций на месте функции:

```
((list + - *) 5 3) → (8 2 15)
```

Применение списка функций возвращает список из результатов применения каждого элемента-функции к соответствующим аргументам. Фактически, предыдущий пример — это краткая запись для

```
(map (lambda (f) (f 5 3))
     (list + - *) )
```

Наконец, мы могли бы разрешить функции быть вторым элементом аппликации, эмулируя привычную инфиксную запись: выражение `(1 + 2)` вернёт `3`. К примеру, такое поведение реализовано в DWIM из [Tei74, Tei76].

Но, во-первых, подобные инновации опасны, так как они уменьшают количество явно ошибочных форм, а значит, затрудняют поиск ошибок, которые иначе бы легко находились. Во-вторых, они не дают какой-либо существенной экономии и в конце концов редко используются. В-третьих, они ещё сильнее разделяют функции и функциональные объекты, которые можно применять как функции. Списки или числа становятся вызываемыми, но вовсе не обязуются быть функциями. В итоге становятся возможными безобразия вроде

```
(apply (list 2 (list 0 (+ 1 2)))
       '(foo bar hux wok) )
  → (hux (foo wok))
```

Поэтому не рекомендуется реализовывать данные инновации как часть базового языка. [**см. упр. 2.3**]

2.4. Сравнение Lisp_1 и Lisp_2

Подходя к окончанию наших исследований Lisp_1 и Lisp_2, что мы можем сказать в итоге об этих двух подходах?

Scheme является Lisp_1, на нём приятно программировать и ему легко обучать, так как процесс вычислений прост и последователен. Lisp_2 же более сложен, потому что существование двух миров требует использования особых форм для перехода между ними. Common Lisp является не совсем Lisp_2, так как имеет множество других пространств имён: для меток лексических переходов, для меток форм `tagbody` и т. д. Поэтому иногда его называют Lisp_n, так как одно и то же имя может вести себя различными способами в зависимости от синтаксической роли. Языки со строгим синтаксисом (или, как говорят, с деспотичным синтаксисом) часто имеют множество пространств имён или множество окружений (для переменных, для функций, для типов и т. д.). Каждое из этих окружений имеет свои особенности. Например, если функциональное окружение неизменяемо (то есть функциям нельзя присваивать новые значения), то становится очень легко оптимизировать вызовы локальных функций.

В программах на Lisp_2 функции чётко отделены от остальных вычислений. Это очень выгодное разграничение, так что, по мнению [Sén89], его следует

применять всем хорошим компиляторам Scheme. Внутри них программы переводятся на Lisp_2-язык, который удаётся лучше компилировать. Компилятор чётко определяет каждое место, куда необходимо вставить `funcall`, — те места, где функции вычисляются динамически. Пользователи Lisp_2 вынуждены делать часть работы компилятора вручную, так что в итоге они лучше понимают стоимость использования своих конструкций.

На предыдущих страницах мы разобрали довольно много вариаций, так что сейчас будет полезным собрать всё воедино и дать ещё одно определение — простейшее из возможных — очередного Lisp_2-языка, похожего на Common Lisp. В этот раз мы ограничимся только введением функции `f.lookup`, которая ищет функцию по имени в функциональном окружении, а если не находит, то вызывает `wrong`. Это позволит гарантировать конечное время выполнения `f.lookup`. Конечно, следствием такого похода будет появление своеобразных ошибок замедленного действия, так как само по себе обращение к неопределённой функции не считается ошибочным. Проблемы возникают лишь при попытке вызова, которая может произойти гораздо позже, а то и вовсе никогда.

```
(define (f.evaluate e env fenv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (f.evaluate (cadr e) env fenv)
                      (f.evaluate (caddr e) env fenv)
                      (f.evaluate (cadddr e) env fenv) ))
        ((begin)  (f.eprogn (cdr e) env fenv))
        ((set!)   (update! (cadr e)
                           env
                           (f.evaluate (caddr e) env fenv) ))
        ((lambda) (f.make-function (cadr e) (cddr e) env fenv))
        ((function)
         (cond ((symbol? (cadr e))
                (f.lookup (cadr e) fenv) )
               ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
                (f.make-function
                  (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
               (else (wrong "Incorrect function" (cadr e))) ) )
        ((flet)
         (f.progn (cddr e)
                  env
                  (extend fenv (map car (cadr e))
                          (map (lambda (def)
```

```
                                  (f.make-function (cadr def)
                                                   (cddr def)
                                                   env fenv ) )
                              (cadr e) ) ) ) )
          ((labels)
           (let ((new-fenv (extend fenv
                                   (map car (cadr e))
                                   (map (lambda (def) 'void)
                                        (cadr e) ) )))
             (for-each (lambda (def)
                         (update! (car def)
                                  new-fenv
                                  (f.make-function (cadr def) (cddr def)
                                                   env new-fenv ) ) )
                       (cadr e) )
             (f.eprogn (cddr e) env new-fenv) ) )
          (else (f.evaluate-application (car e)
                                        (f.evlis (cdr e) env fenv)
                                        env
                                        fenv )) ) ) )

(define (f.evaluate-application fn args env fenv)
  (cond ((symbol? fn)
         ((f.lookup fn fenv) args) )
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (f.eprogn (cddr fn)
                   (extend env (cadr fn) args)
                   fenv ) )
        (else (wrong "Incorrect functional term" fn)) ) )

(define (f.lookup id fenv)
  (if (pair? fenv)
      (if (eq? (caar fenv) id)
          (cdar fenv)
          (f.lookup id (cdr fenv)) )
      (lambda (values)
        (wrong "No such functional binding" id) ) ) )
```

Ещё одно важное, по мнению [GP88], практическое различие между Lisp_1 и Lisp_2 состоит в читабельности. Хотя, конечно, опытные программисты вряд ли будут писать что-то вроде:

```
(defun foo (list)
  (list list) )
```

С точки зрения Lisp_1, `(list list)` — это вполне допустимое самоприменение,[2]

[2] Существуют и другие самоприменения, которые имеют смысл, хотя их и не особо много. Например, `(number? number?)`.

но в Lisp_2 это выражение имеет совершенно различный смысл. В Common Lisp эти два имени принадлежат различным окружениям и не конфликтуют. Тем не менее, лучше избегать подобного стиля именования и не называть локальные переменные именами известных глобальных функций; это поможет избежать проблем с макросами и сделает программы менее зависимыми от используемого диалекта.

Следующее отличие между Lisp_1 и Lisp_2 заключается в собственно макросах. В Common Lisp довольно непросто написать макрос, раскрывающийся в `lambda`-форму, который был бы полезен, например, при реализации объектной системы. Дело в том, что Common Lisp ограничивает места, где может появляться `lambda`. Она может стоять только на месте функции, так что конструкция `(... (lambda ...) ...)` вызовет ошибку в Common Lisp. Единственное исключение — `lambda` может быть первым аргументом `function`. Но сама `function` может быть только параметром функции, так что `((function (lambda ...)) ...)` — это снова ошибка. Если макрос не знает, где именно он раскроется: как аргумент или как функция, то его нельзя использовать без некоторого усложнения программ. Для той же системы объектов, например, придётся раскрывать макрос в `(function (lambda ...))`, а потом вручную оборачивать его в `funcall` там, где он вызывается как функция.

Наконец, стоит упомянуть радикальное решение, используемое многими языками. Можно запретить использование одного и того же имени даже в различных окружениях. Предыдущий пример с `list` вызвал бы ошибку, так как имя `list` уже используется в функциональном окружении. Практически все реализации Лиспа и Scheme запрещают переменной иметь то же имя, что и функция или макрос. Такое правило действительно облегчает жизнь.

2.5. Пространства имён

Окружения устанавливают соответствия между именами и объектами. На данный момент нам известны два типа окружений: обычное окружение `env` и функциональное `fenv`. Причина, по которой они разделены, состоит в желании ускорить вызовы функций и отделить функции от переменных. Но это разделение потребовало от нас ввода двух различных вычислителей, а также механизма переноса сущностей одного окружения в другое, — и это усложнило язык. При обсуждении динамических переменных мы упомянули, что современные диалекты Лиспа (вроде ILOG Talk, EuLisp, ISLisp) помещают динамические переменные в отдельное пространство имён. Сейчас мы подробнее рассмотрим это решение. Занимаясь этим, мы проиллюстрируем саму идею *пространств имён*.

Окружения можно понимать как абстрактный тип данных. Они содержат *привязки* (bindings) объектов к их именам. Объекты могут быть как значениями (полноценными объектами первого класса, которые можно передавать,

копировать, присваивать и т. д.), так и сущностями (объектами второго класса, которыми можно оперировать лишь посредством их имён и, обычно, только с помощью ограниченного числа специальных форм или иных синтаксических конструкций). В данный момент нам известен только один тип подобных сущностей — сами привязки. Они существуют лишь потому, что они — это нечто, захватываемое замыканиями. Мы рассмотрим их свойства подробнее, когда будем изучать побочные эффекты.

Есть множество способов использования окружений. Нас может интересовать, присутствует ли какое-то имя в окружении; мы можем искать объект по его имени; мы можем искать саму привязку, чтобы изменить её. Мы также можем изменять окружение, добавляя в него новые привязки, будь это текущее, локальное или глобальное окружение. Конечно же, не все возможности необходимы для каждого из окружений. В действительности, многие окружения полезны именно благодаря накладываемым ими ограничениям. Следующая таблица показывает особенности окружения переменных Scheme:

Ссылка	x
Значение	x
Изменение	`(set!` x `...)`
Расширение	`(lambda (...` x `...) ...)`
Определение	`(define` x `...)`

Мы будем использовать такие таблицы и понятия из них довольно часто при обсуждении свойств окружений, так что остановимся на них подробнее. Первая строка показывает синтаксис, используемый для получения ссылки на переменную, обращения к ней. Вторая строка показывает синтаксис, используемый для получения значения переменной. В данном случае они совпадают, но так бывает отнюдь не всегда. Третья строка показывает, как связать переменную с другим значением. Четвёртая строка показывает, как расширить окружение локальных переменных новой привязкой: с помощью `lambda`-формы или, конечно же, макросов вроде `let` или `let*`, раскрывающихся в `lambda`-формы. Наконец, последняя строка показывает, как определить глобальную переменную. Не волнуйтесь, если эти различия сейчас для вас кажутся чересчур тонкими и излишними, мы и в дальнейшем будем использовать подобные таблицы для пояснения вариантов использования переменных, а там различия будут более заметными.

Например, пространство имён функций Lisp_2, рассмотренного в начале главы, описывается следующей таблицей:

Ссылка	`(`f `...)`
Значение	`(function` f`)`
Изменение	запрещено
Расширение	`(flet (...` `(`f `...) ...) ...)`
Определение	не рассматривалось ранее (`defun`)

2.5.1. Динамические переменные

Динамические переменные принципиально отличаются от лексических, поэтому имеет смысл отличать их также и на уровне окружений. Следующая таблица показывает желаемые свойства нашего нового окружения, окружения динамических переменных:

Ссылка	не может быть получена
Значение	`(dynamic` *d*`)`
Изменение	`(dynamic-set!` *d* `...)`
Расширение	`(dynamic-let (...` `(`*d* `...) ...) ...)`
Определение	здесь не рассматривается

Окружение учитывает новые особенности: можно создавать локальные динамические переменные[3] с помощью формы `dynamic-let`, подобной `let` и `flet`; значение динамической переменной получается с помощью `dynamic`, а изменяется — с помощью `dynamic-set!`.

Пока что это три специальные формы, но далее будут рассмотрены другие варианты реализации. А сейчас мы всего лишь добавим в наш интерпретатор `f.evaluate` поддержку динамического окружения: `denv`. Это окружение будет содержать только динамические переменные. Новый интерпретатор, назовём его Lisp_3, будет использовать функции с префиксом `df.`, чтобы не путать их с остальными. Вот его код. (Форма `flet` не показана для краткости.)

```
(define (df.evaluate e env fenv denv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (df.evaluate (cadr e) env fenv denv)
                      (df.evaluate (caddr e) env fenv denv)
                      (df.evaluate (cadddr e) env fenv denv) ))
        ((begin)  (df.eprogn (cdr e) env fenv denv))
        ((set!)   (update! (cadr e)
                           env
                           (df.evaluate (caddr e) env fenv denv) ))
        ((function)
         (cond ((symbol? (cadr e))
                (f.lookup (cadr e) fenv) )
               ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
```

[3] Название «локальные переменные» не совсем удачно, так как поведение динамических переменных кардинально отличается от поведения обычных (лексических) переменных.

```
                (df.make-function
                 (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
                (else (wrong "Incorrect function" (cadr e))) ) )
          ((dynamic) (lookup (cadr e) denv))
          ((dynamic-set!)
           (update! (cadr e)
                    denv
                    (df.evaluate (caddr e) env fenv denv) ) )
          ((dynamic-let)
           (df.eprogn (cddr e)
                      env
                      fenv
                      (extend denv
                              (map car (cadr e))
                              (map (lambda (e)
                                     (df.evaluate e env fenv denv) )
                                   (map cadr (cadr e)) ) ) ) )
          (else (df.evaluate-application (car e)
                                         (df.evlis (cdr e) env fenv denv)
                                         env
                                         fenv
                                         denv )) ) ) )

(define (df.evaluate-application fn args env fenv denv)
  (cond ((symbol? fn) ((f.lookup fn fenv) args denv))
        ((and (pair? fn) (eq? (car fn) 'lambda))
         (df.eprogn (cddr fn)
                    (extend env (cadr fn) args)
                    fenv
                    denv ) )
        (else (wrong "Incorrect functional term" fn)) ) )

(define (df.make-function variables body env fenv)
  (lambda (values denv)
    (df.eprogn body (extend env variables values) fenv denv) ) )

(define (df.eprogn e* env fenv denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (begin (df.evaluate (car e*) env fenv denv)
                 (df.eprogn (cdr e*) env fenv denv) )
          (df.evaluate (car e*) env fenv denv) )
      empty-begin ) )
```

Для поддержки нового окружения, `denv`, потребовалось изменить прототипы `df.evaluate` и `df.eprogn`, чтобы не терять это окружение при вычислениях. Далее, `df.evaluate` определяет три новые специальные формы для

операций над `denv`, динамическим окружением. Есть и менее заметные изменения: `df.evaluate-application` передаёт в функции *текущее* динамическое окружение. Мы уже встречались с таким поведением [**см. стр. 38**], когда были вынуждены передавать текущее лексическое окружение в вызываемую функцию.

При применении функции используются одновременно несколько окружений. Есть окружение с переменными и функциями, захваченными при определении функции. Есть также окружение с динамическими переменными, существующими в момент вызова. Это окружение не может быть захвачено и сохранено в замыкании, каждый раз значение динамической переменной ищется заново в текущем динамическом окружении. Оно может там отсутствовать, что, конечно же, приводит к ошибке. Возможны и другие варианты реализации: например, сделать единое глобальное окружение динамических переменных, как в ISLisp; или выдать каждому модулю собственное динамическое окружение, как в EuLisp; или даже иметь глобальное окружение лексических переменных, как в Common Lisp. [**см. стр. 70**]

Одно из преимуществ отдельного окружения: становится чётко видно, какие переменные динамические, а какие нет. При всяком обращении к динамическому окружению необходимо использовать специальную форму с префиксом `dynamic`. Явные обращения к динамическим переменным невозможно не заметить. Это очень важно, так как в Lisp_3 поведение функции определяется не только значениями локальных переменных, но и текущим состоянием динамического окружения.

Среди традиционных вариантов использования динамических окружений наиболее полезным является обработка ошибок. При возникновении ошибки или исключительной ситуации создаётся некий объект, описывающий, что произошло, и к этому объекту применяется соответствующая функция, которая обработает исключительную ситуацию (и, возможно, попытается восстановить работоспособность программы). Этот обработчик мог бы быть общей глобальной функцией, но это потребовало бы кучи ненужных присваиваний для использования правильного обработчика в правильное время. «Герметичность» лексических окружений плохо сочетается со сквозным указанием различных функций-обработчиков. Хотя мы всегда можем ограничить время жизни локальных переменных, заключив вычисления в `let` или `dynamic-let`, но у `dynamic-let` есть несколько серьёзных преимуществ:

1) создаваемые привязки не могут быть захвачены;

2) эти привязки доступны только во время вычислений, вложенных в форму;

3) привязки автоматически уничтожаются после завершения вычислений.

Поэтому `dynamic-let` идеально подходит для временной установки функций-обработчиков ошибок.

Вот ещё один пример разумного использования динамических переменных. Функции вывода в Common Lisp настраиваются динамическими переменными вроде `*print-base*`, `*print-circle*` и т. д. В них хранится информация об основании счисления для вывода чисел, о том, могут ли выводимые данные содержать циклические списки, и тому подобное. Конечно, можно передавать всю эту информацию через аргументы, но только представьте, что вместо (`print` *выражение*) приходилось бы писать (`print` *выражение escape-символы? основание циклы? pretty-print? регистр? уровень-вложенности векторы? использовать-gensym?*). Каждый раз. Динамические переменные позволяют один раз установить значения по умолчанию для таких параметров и больше никогда их не указывать, если не требуется особого поведения.

Scheme использует похожий механизм для указания портов ввода-вывода. Можно написать (`display` *выражение*) или[4] (`display` *выражение порт*). Первая форма, с одним аргументом, выводит *выражение* в текущий порт вывода. Вторая же использует явно указанный порт. Функция `with-output-to-file` позволяет указать порт вывода, который будет текущим на время вычисления выражения. Узнать текущий порт можно с помощью `current-output-port`. Вот так определяется функция,[5] которая выводит циклические списки в текущий порт:

```
(define (display-cyclic-spine list)
  (define (scan l1 l2 flip)
    (cond ((atom? l1)  (unless (null? l1)
                          (display " . ") (display l1) )
                       (display ")") )
          ((eq? l1 l2) (display "...)"))
          (else        (display (car l1))
                       (when (pair? (cdr l1)) (display " "))
                       (scan (cdr l1)
                             (if (and flip (pair? l2))
                                 (cdr l2)
                                 l2 )
                             (not flip) ) ) ) )
  (display "(")
  (scan list (cons 42 list) #f) )

(display-cyclic-spine        ; напечатает (1 2 3 4 1 ...)
  (let ((l (list 1 2 3 4)))
    (set-cdr! (cdddr l) l)
    l ) )
```

[4] Это две различные функции `display`, принимающие один и два аргумента соответственно. В Scheme нет поддержки опциональных аргументов и значений по умолчанию на уровне языка. Лисп же поддерживает эту идею, поэтому не нуждается в динамических переменных для реализации подобного поведения.

[5] См. также реализацию функции `list-length` из Common Lisp.

Можно даже составить таблицу характеристик для портов вывода Scheme:

Ссылка	автоматически, если не упоминается явно
Значение	`(current-output-port)`
Изменение	запрещено
Расширение	`(with-output-to-file` *имя-файла замыкание*`)`
Определение	неприменимо

В Common Lisp данный механизм явно использует динамические переменные. По умолчанию функции вывода вроде `print` или `write` используют порт, хранящийся в переменной `*standard-output*`.[6] Мы можем проэмулировать[7] `with-output-to-file` следующим образом:

```
(define (with-output-to-file filename thunk)
  (dynamic-let ((*standard-output* (open-input-file filename)))
    (thunk) ) )
```

2.5.2. Динамические переменные в Common Lisp

Хотя Common Lisp разделяет динамические и лексические переменные с точки зрения вычислений, но синтаксис их использования отличается слабо. В нём нет формы `dynamic-let`, но её можно легко сымитировать:

```
(dynamic-let ((x α)) ≡ (let ((x α))
  β )                    (declare (special x))
                         β )
```

Отличие состоит в том, что для получения значения динамической переменной `x` внутри β нет надобности использовать форму `dynamic`, достаточно писать просто `x`. Причина такого поведения в том, что выражение `(declare (special x))` означает сразу две вещи: привязка, устанавливаемая `let` для `x` должна быть динамической, а каждая ссылка на `x` внутри тела `let` должно пониматься как `(dynamic x)`.

Это не совсем удобно, так как внутри β нельзя обращаться к лексической переменной `x`, нам будет видна только её динамическая тёзка. Можно было бы пойти другим путём, указывая конкретные места, где необходимо использовать динамическую переменную `x` с помощью конструкции `(locally (declare (special x)) x)`. Это в точности идентично нашей форме `dynamic`.

Стратегия Common Lisp состоит в указании типа привязки с помощью конструкций языка. Мы можем реализовать этот механизм, сделав следующие изменения в интерпретаторе:

[6] По соглашению, имена динамических переменных окружаются звёздочками, чтобы выделить их.

[7] Разумеется, это не Common Lisp. Это наш Lisp_3, определённый чуть выше.

```
(define (df.evaluate e env fenv denv)
  (if (atom? e)
      (cond ((symbol? e) (cl.lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (df.evaluate (cadr e) env fenv denv)
                      (df.evaluate (caddr e) env fenv denv)
                      (df.evaluate (cadddr e) env fenv denv) ))
        ((begin)  (df.eprogn (cdr e) env fenv denv))
        ((set!)   (update! (cadr e)
                           env
                           (df.evaluate (caddr e) env fenv denv) ))
        ((function)
         (cond ((symbol? (cadr e))
                (f.lookup (cadr e) fenv) )
               ((and (pair? (cadr e)) (eq? (car (cadr e)) 'lambda))
                (df.make-function
                 (cadr (cadr e)) (cddr (cadr e)) env fenv ) )
               (else (wrong "Incorrect function" (cadr e))) ) )
        ((dynamic) (lookup (cadr e) denv))
        ((dynamic-set!)
         (update! (cadr e)
                  denv
                  (df.evaluate (caddr e) env fenv denv) ) )
        ((dynamic-let)
         (df.eprogn (cddr e)
                    (special-extend env
                                    (map car (cadr e)) )
                    fenv
                    (extend denv
                            (map car (cadr e))
                            (map (lambda (e)
                                   (df.evaluate e env fenv denv) )
                                 (map cadr (cadr e)) ) ) ) )
        (else (df.evaluate-application (car e)
                                       (df.evlis (cdr e) env fenv denv)
                                       env
                                       fenv
                                       denv )) ) ) )

(define (special-extend env variables)
  (append variables env) )
```

```
(define (cl.lookup var env denv)
  (let look ((env env))
    (if (pair? env)
        (if (pair? (car env))
            (if (eq? (caar env) var)
                (cdar env)
                (look (cdr env)) )
            (if (eq? (car env) var)
                ;; ищем в текущем динамическом окружении
                (let lookup-in-denv ((denv denv))
                  (if (pair? denv)
                      (if (eq? (caar denv) var)
                          (cdar denv)
                          (lookup-in-denv (cdr denv)) )
                      ;; если не находим, ищем в глобальном лексическом
                      (lookup var env.global) ) )
                (look (cdr env)) ) )
        (wrong "No such binding" var) ) ) )
```

Теперь разберём, как это работает. Когда `dynamic-let` создаёт динамическую переменная, она не только связывает её со значением в динамическом окружении, но и помечает её как динамическую в лексическом окружении, записывая туда её имя. Для поддержки этих меток изменяется механизм поиска значения по ссылке (`cl.lookup`): он должен проанализировать ссылку, чтобы определить тип привязки (лексическая или динамическая), после чего отыскать значение в правильном окружении. Также, если переменная не найдена в текущем динамическом окружении, то следующим просматривается глобальное лексическое окружение, которое в Common Lisp одновременно является глобальным динамическим.

Приведём пример работы такого Lisp_3, имитирующего Common Lisp:

```
(dynamic-let ((x 2))
  (+ x                          ; динамический x
     (let ((x (+                ; лексический
                 x x )))        ; динамические
       (+ x                     ; лексический
          (dynamic x) ) ) ) )   ; динамический
→ 8
```

2.5.3. Динамические переменные без специальных форм

Сейчас для работы с динамическими переменными используются целых три специальные формы. Так как Scheme исповедует минимализм по отношению к количеству специальных форм, стоит подумать о других вариан-

тах. Не будем рассматривать их все, а остановимся на следующем, так как он использует всего две функции. Первая функция динамически связывает два значения; вторая функция может по первому значению найти второе. На роль идентификаторов динамических переменных прекрасно подходят символы. Кроме них нам потребуется некий изменяемый тип данных вроде точечных пар, если мы хотим изменять установленные связи. Так наше решение будет удовлетворять аскетичным традициям Scheme.

Во время изучения данного подхода мы будем использовать интерпретатор с двумя окружениями: `env` и `denv`. Это тот же предыдущий интерпретатор, только из него убрано несколько вещей: ненужные специальные формы, поддержка функциональных окружений и ссылки на переменные как в Common Lisp. Остались только мы и динамические окружения. Такой интерпретатор несколько менее полезен, так как динамическое окружение уже нельзя расширять непосредственно, но, тем не менее, это окружение всё ещё передаётся в каждую функцию — и этого достаточно! Чтобы отличать эту вариацию от предыдущих, её функциям выдан префикс `dd`.

```
(define (dd.evaluate e env denv)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote)  (cadr e))
        ((if)     (if (dd.evaluate (cadr e) env denv)
                      (dd.evaluate (caddr e) env denv)
                      (dd.evaluate (cadddr e) env denv) ))
        ((begin)  (dd.eprogn (cdr e) env denv))
        ((set!)   (update! (cadr e) env
                           (dd.evaluate (caddr e) env denv) ))
        ((lambda) (dd.make-function (cadr e) (cddr e) env))
        (else (invoke (dd.evaluate (car e) env denv)
                      (dd.evlis (cdr e) env denv)
                      denv )) ) ) )

(define (dd.make-function variables body env)
  (lambda (values denv)
    (dd.eprogn body (extend env variables values) denv) ) )

(define (dd.evlis e* env denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (cons (dd.evaluate (car e*) env denv)
                (dd.evlis (cdr e*) env denv) )
```

```
            (list (dd.evaluate (car e*) env denv)) )
        '() ) )

(define (dd.eprogn e* env denv)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (begin (dd.evaluate (car e*) env denv)
                 (dd.eprogn (cdr e*) env denv) )
          (dd.evaluate (car e*) env denv) )
      empty-begin ) )
```

Как мы и обещали, теперь определим две функции. Первую назовём `bind-with-dynamic-extent`; это длинное название, так что сократим его до `bind/de`. Первым её аргументом является ключ `tag`; вторым — значение `value`, связываемое с ключом; третьим — замыкание `thunk`, функция без аргументов, которая будет вызвана в расширенном динамическом окружении.

```
(definitial bind/de
  (lambda (values denv)
    (if (= 3 (length values))
        (let ((tag   (car values))
              (value (cadr values))
              (thunk (caddr values)) )
          (invoke thunk '()
                  (extend denv (list tag) (list value)) ) )
        (wrong "Incorrect arity" 'bind/de) ) ) )
```

Следующая функция будет использовать динамическое окружение. Так как нам надо что-то делать в случае, если динамической переменной с запрошенным именем нет, то функция `assoc/de` первым аргументом принимает ключ, а вторым — функцию. Она вызовет полученную функцию и передаст ей ключ, если не найдёт его в динамическом окружении.

```
(definitial assoc/de
  (lambda (values current.denv)
    (if (= 2 (length values))
        (let ((tag     (car values))
              (default (cadr values)) )
          (let look ((denv current.denv))
            (if (pair? denv)
                (if (eqv? tag (caar denv))
                    (cdar denv)
                    (look (cdr denv)) )
                (invoke default (list tag) current.denv) ) ) )
        (wrong "Incorrect arity" 'assoc/de) ) ) )
```

Можно реализовать несколько вариантов её поведения в зависимости от используемого механизма сравнения (`eqv?` или `equal?`). [**см. упр. 2.4**]

Перепишем предыдущий пример:

```
(bind/de 'x 2
  (lambda () (+ (assoc/de 'x error)
                (let ((x (+ (assoc/de 'x error)
                            (assoc/de 'x error) )))
                  (+ x (assoc/de 'x error)) ) )) )
→ 8
```

Таким образом, мы опровергли необходимость использования специальных форм для реализации механизма динамических переменных. Заодно мы получили возможность связывать что угодно с чем угодно. Конечно, это преимущество — ничто по сравнению с тем, что есть гораздо более эффективные реализации динамических переменных (даже без учёта многопоточности). Хотя бы то же ближнее связывание, которое лишь требует, чтобы ключ был символом.[8] С другой стороны, в данном варианте не потеряна ссылочная прозрачность. Но всё равно доступ к динамическим переменным требует недешёвых вызовов функций. Наше решение ещё довольно далеко от органичного сочетания динамических и лексических переменных в Common Lisp.

Среди всех неудобств стоит отметить ещё то, что использование `bind/de` требует использования `assoc/de` и написания функции-обработчика. Хотя, естественно, с помощью макросов это можно спрятать. Другое неудобство возникает уже для компилятора: ведь ему надо будет генерировать код для создания и использования динамических переменных. К счастью, они помечены вызовами соответствующих функций, так что дальнейшее отдаётся на откуп компилятору: генерировать код в лоб (это легче) или вставить свою, более эффективную реализацию.

2.5.4. В заключение о пространствах имён

После отступления к динамическим переменным, вернёмся к идее пространств имён в общем: специализированные окружения для специализированных объектов. Мы уже видели Lisp_3 в деле, а также разобрали механизм динамических переменных в Common Lisp.

Тем не менее, наша последняя реализация — та, что с двумя функциями вместо трёх специальных форм, — поднимает каверзный вопрос. Если это Lisp_n, то чему равно n? Базируется она на Scheme, но всё же явно имеет два окружения: `env` и `denv`. В то же время, вычислитель у неё лишь один, что является отличительной чертой Scheme и класса Lisp_1. Однако, нам пришлось довольно сильно модифицировать интерпретатор (просто сравните `evaluate` и `dd.evaluate`), чтобы реализовать функции `bind/de` и `assoc/de`. Мы столкнулись с примитивными функциями, которые нельзя выразить тем же языком, если только они уже не определены; более того, само существование этих

[8] Многие реализации Лиспа не считают ключевые слова вроде `nil` или `if` символами (а значит, и легальными именами переменных).

функций глубоко влияет на процесс вычислений. В следующей главе будет такая же ситуация с `call/cc`.

Короче говоря, похоже, что у нас получился Lisp_1, если смотреть на количество вычислителей, и Lisp_2 — если смотреть на пространства имён. Обобщением этих парадоксов является мнение, что наличие списка свойств у символов является чертой Lisp_n, где n может быть произвольным. [**см. упр. 2.6**] Так как наши пространства имён объективно существуют, а значения соответствующих переменных вычисляются особым образом (пусть и с помощью примитивных функций, а не специальных форм), то будем считать нашу реализацию представителем класса Lisp_2.

Остаётся ещё один урок, который можно извлечь из рассмотрения лексических и динамических переменных. Common Lisp старается унифицировать доступ к переменным из различных пространств имён, предоставляя одинаковый синтаксис. Поэтому необходимо знать правила, по которым он определяет, из какого пространства имён взять переменную. К сожалению, они не всегда однозначны; например, Common Lisp не различает глобальное динамическое и глобальное лексическое окружения. Далее, в Lisp 1.5 существовала концепция констант, определяемых специальной формой `csetq` (`setq` для констант).

Ссылка	x
Значение	x
Изменение	`(csetq` x *форма*`)`
Расширение	запрещено
Определение	`(csetq` x *форма*`)`

Введение констант тоже делает синтаксис неоднозначным. Когда мы пишем `foo` — это может быть как константа, так и переменная. Правило разрешения противоречий в Lisp 1.5 было таково: если существует константа с именем `foo`, то вернуть её значение; иначе искать одноимённую переменную в лексическом пространстве имён. Но: «константы» можно изменять (представьте себе!) с помощью той же формы `csetq`, что используется для их создания. Таким образом, константы Lisp 1.5 соответствуют глобальным переменным Scheme, только с обратным приоритетом: в Scheme сначала ищется локальная лексическая переменная, а глобальная переменная является лишь значением по умолчанию на случай, если локальная переменная не будет найдена.

Проблема имеет довольно общий характер. Если для доступа к нескольким пространствам имён используется одинаковый синтаксис, то необходимы чётко прописанные правила разрешения неоднозначностей.

2.6. Рекурсия

Рекурсия естественна для Лиспа, но мы пока так ничего и не сказали о том, как же она реализуется. Далее мы проанализируем различные типы рекурсии и вызываемые ими проблемы.

2.6.1. Простая рекурсия

Наверное, наиболее известной простой рекурсивной функцией является факториал, определяемый следующим образом:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

Язык, который был определён в предыдущей главе, не знает, что такое `define`, так что давайте представим, что это макрос, который раскрывается в следующий код:

```
(set! fact (lambda (n)
             (if (= n 0) 1
                 (* n (fact (- n 1))) ) ))
```

Здесь мы видим присваивание, изменение значения переменной `fact`. Это изменение бессмысленно, если переменная `fact` не существует. Глобальное окружение можно считать местом, где уже существуют все возможные переменные. В этой *виртуальной реальности* интерпретатор (а точнее, его часть, ответственная за чтение программ) должен создать привязку для переменной, когда впервые видит её имя, после чего невозмутимо продолжить работу, будто бы эта переменная уже сто лет как здесь. Каждая переменная уже существует до своего первого использования, а значит, её определение — это лишь изменение значения существующей переменной. Но при таком подходе возникает проблема с получением значения переменной, которой ещё ничего не было присвоено. Интерпретатор должен отловить эту ошибку, когда переменная вроде как есть, но значения у неё ещё нет. Как видите, идея привязок не так уж и проста, мы рассмотрим её подробнее в четвёртой главе. [**см. стр. 142**]

Можно избавиться от проблем с привязками, которые существуют, но еще не инициализированы, если принять другую позицию. Пусть изменять можно только существующие переменные; иными словами, переменной нет, если она не была определена явно. Для этого используется специальная форма `define`, создающая переменные. Без неё у нас не получится ни обратиться к переменной, ни установить её значение. Однако, посмотрим теперь на это с другой стороны:

```
(define (display-pi)
  (display pi) )
(define pi 2.7182818285)      ; ой, не та константа
```

```
(define (print-pi)
  (display pi) )
(define pi 3.1415926536)
```

Допустимо ли такое определение `display-pi`? Её тело ссылается на `pi`, которая ещё не определена. Четвёртый `define` исправляет ошибку во втором; но даже если `define` создаёт новую привязку, можно ли создать ещё одну с таким же именем?

Эти вопросы не имеют однозначного ответа. Есть как минимум два варианта: считать глобальные определения лексическими (как это делает ML) или считать их динамическими (как это принято в Лиспе).

В глобальном окружении, которое *полностью лексично* — будем называть его *гиперстатическим*, — мы не можем использовать переменную (ссылаться на неё, получать или изменять её значение), если она не существует. В нём определение функции `display-pi` будет ошибочным, так как оно ссылается на переменную `pi`, которая не существует на момент определения. Любой вызов функции `display-pi` привёл бы к ошибке «Неизвестная переменная: `pi`» (если бы это не случилось ещё при её определении). Но с функцией `print-pi` всё в порядке, она будет выводить значение, существовавшее на момент её определения (в данном случае это `2.7182818285`), и ничто не сможет изменить выводимое значение. Здесь переопределение `pi` вполне допустимо и создаст новую переменную `pi`, которая будет использоваться в последующих выражениях в пределах своей области видимости. Предыдущий пример можно представить примерно таким образом:

```
(let ((display-pi (lambda () (display pi))))
  (let ((pi 2.7182818285))
    (let ((print-pi (lambda () (display pi))))
      (let ((pi 3.1415926536))
        ... ) ) ) )
```

В Лиспе, как было сказано, принят динамический подход. Предполагается, что может существовать максимум одна глобальная переменная с уникальным именем, и эта переменная видна везде, в том числе в своём собственном определении. Лисп поддерживает опережающие ссылки без какого-либо специального синтаксиса. (Под таким синтаксисом понимаются явные объявления вроде ключевого слова `forward` Паскаля или прототипов ISO C [ISO90].)

Выбор отнюдь не так прост. Возвращаясь к факториалу; в окружении, где будет вычисляться `lambda`-форма, необходимо ответить на вопрос: «Чему здесь равно значение `fact`?» Если вычисление происходит в глобальном окружении, где нет привязки для `fact`, то эта функция в принципе не может быть рекурсивной. Причина: функция ссылается на некую переменную `fact`, значение которой следует искать в окружении, захваченном замыканием (в том, которое состоит из свободных переменных и параметров функции). Так как в момент создания замыкания переменная `fact` не существовала, то и в окружении её нет. Следовательно, для рекурсии её необходимо каким-то образом

туда добавить. Проще всего это сделать, выбрав динамический подход к глобальному окружению, который снимает вопрос существования переменных в принципе. При гиперстатическом же подходе надо убедиться в том, что `define` создаёт привязку для `fact` перед тем, как вычислять замыкание, которое станет значением `fact`. Резюмируя: простая рекурсия требует глобального окружения.

2.6.2. Взаимная рекурсия

Теперь предположим, что мы хотим определить две взаимно рекурсивные функции. Возьмём для примера `odd?` и `even?`, реализующие (весьма медленную) проверку натуральных чисел на чётность. Они определяются следующим образом:

```
(define (even? n)
  (if (= n 0) #t (odd? (- n 1))) )
(define (odd? n)
  (if (= n 0) #f (even? (- n 1))) )
```

Можно менять их местами, но в любом случае первое определение не будет знать о втором; в данном случае `even?` не знает в момент определения про `odd?`. И опять, кажется, решением будет глобальное окружение с заранее созданными переменными: оба замыкания захватывают глобальное окружение, в котором есть все возможные переменные, среди них, в частности, и необходимые `odd?` и `even?`. Конечно, мы пока оставим в стороне вопрос, как именно реализуется захват только необходимых привязок.

Довольно непросто перенести это поведение в мир с гиперстатическим глобальным окружением, так как здесь уж точно первое определение никогда не сможет узнать о втором. Одно из решений состоит в том, чтобы определять эти две функции одновременно, тогда не будет никаких первых и вторых, и обе функции смогут ссылаться друг на друга без проблем. (Мы вернёмся к этому вопросу чуть позже, после изучения локальной рекурсии.) Например, когда-то в Lisp 1.5 имелась форма `define` с подобной возможностью:

```
(define ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))))
```

Таким образом, с помощью глобального окружения и некоторых ухищрений можно выразить и взаимную рекурсию.

Но что случится, если нам понадобятся локальные рекурсивные функции?

2.6.3. Локальная рекурсия в Lisp_2

Некоторые проблемы, с которыми мы столкнулись при попытках определить `fact` в глобальном окружении, возвращаются при попытках определить `fact` локально. Нам надо сделать что-то, чтобы вызов `fact` из тела `fact` был рекурсивным. Чтобы `fact` из функционального окружения была связана

с функцией вычисления факториала даже внутри самой функции вычисления факториала. И вот здесь как раз проявляется отличие между локальным и глобальным окружениями: вспомните, что происходит, если искомая локальная переменная не существует. Рассмотрим следующую программу на Lisp_2:

```
(flet ((fact (n) (if (= n 0) 1
                     (* n (fact (- n 1))) )))
  (fact 6) )
```

Форма `flet` связывает функцию вычисления факториала с именем `fact` в своём внутреннем функциональном окружении. Замыкание захватывает функциональное и параметрическое окружения, локальные для формы `flet`. Таким образом, `fact` внутри `fact` ссылается не на функцию `fact`, локальную для тела `flet`, а на какую-то другую функцию `fact` из окружения всей формы `flet`. Эта функция не обязана вычислять факториал (а то и вовсе не существует), так что рекурсию мы не получаем.

Эта проблема была очевидна ещё во времена Lisp 1.5. Для её решения была введена специальная форма `label`, позволявшая определять локальные рекурсивные функции. Пример с факториалом тогда записывается так:

```
(label fact (lambda (n) (if (= n 0) 1
                            (* n (fact (- n 1))) )))
```

Эта форма возвращает анонимную функцию, вычисляющую факториал. Более того, это именно та функция, которая связана с `fact` в своём же теле.

К сожалению, нельзя сказать, что Lisp 1.5 был Lisp_2, а `label`, какой бы удобной она не была, не может легко справиться со взаимной рекурсией. Поэтому гораздо позже, судя по [HS75], был изобретён её n-арный аналог: `labels`. Эта форма имеет тот же синтаксис, что и `flet`, но гарантирует, что замыкания будут создаваться в окружении, где можно ссылаться на локальные функции. С её помощью можно определить и `fact`, и взаимно рекурсивные `odd?` и `even?`:

```
(labels ((fact (n) (if (= n 0) 1
                       (* n (fact (- n 1))) )))
  (fact 6) ) → 720

(funcall (labels ((even? (n) (if (= n 0) #t (odd? (- n 1))))
                  (odd? (n) (if (= n 0) #f (even? (- n 1)))) )
           (function even?) )
         4 ) → #t
```

Так что в Lisp_2 мы имеем две формы для расширения локального функционального окружения: `flet` и `labels`.

2.6.4. Локальная рекурсия в Lisp_1

Проблема определения локальных рекурсивных функций существует и в Lisp_1; решается она похожим способом. Форма `letrec` (рекурсивная `let`) очень похожа по смыслу на `labels`.

В Scheme `let` имеет следующий синтаксис:

```
(let ((переменная1 выражение1)
      (переменная2 выражение2)
       ...
      (переменнаяn выражениеn) )
  выражения... )
```

И она эквивалентна такому выражению:

```
((lambda (переменная1 переменная2 ... переменнаяn) выражения...)
 выражение1 выражение2 ... выражениеn )
```

Поясним, что здесь происходит. Сперва вычисляются все аргументы аппликации: *выражение*$_1$, *выражение*$_2$, . . . , *выражение*$_n$; затем переменные *переменная*$_1$, *переменная*$_2$, . . . , *переменная*$_n$ связываются с только что полученными значениями; наконец, *выражения*, составляющие тело `let`, вычисляются в расширенном окружении внутри неявной формы `begin`, а её значение становится значением всей формы `let`.

Как видим, в принципе нет необходимости делать `let` специальной формой, так как её полностью заменяет `lambda`; следовательно, `let` может быть всего лишь макросом. (Именно так и поступили в Scheme: `let` — это встроенный макрос.) Тем не менее, `let` хороша с точки зрения стиля кодирования, потому что позволяет не разделять имя переменной и её начальное значение большим куском кода. Теперь самое время заметить, что начальные значения локальных переменных формы `let` вычисляются в текущем окружении; в расширенном вычисляется только её тело.

По тем же причинам, с которыми мы столкнулись в Lisp$_2$, это значительно усложняет написание взаимно рекурсивных функций. Поэтому вводится форма `letrec`, аналог `labels`.

Синтаксис `letrec` такой же, как и у `let`. Например:

```
(letrec ((even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
         (odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))) )
  (even? 4) )
```

Отличается `letrec` от `let` тем, что выражения-инициализаторы вычисляются в том же окружении, что и тело `letrec`. Операции, которые выполняет `letrec`, те же, что и у `let`, но их порядок несколько иной. Сначала локальное окружение расширяется переменными `letrec`. Затем в этом расширенном окружении вычисляются начальные значения переменных. Наконец, в том же расширенном окружении вычисляется тело `letrec`. По этому описанию довольно легко понять, как реализовать такое поведение. Действительно, достаточно написать следующее:

```
(let ((even? 'void) (odd? 'void))
  (set! even? (lambda (n) (if (= n 0) #t (odd? (- n 1)))))
  (set! odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
  (even? 4) )
```

Сначала создаются привязки для `even?` и `odd?`. (Их начальные значения не важны, просто `let` и `lambda` требуют какое-то значение.) Затем эти переменные инициализируются значениями, вычисленными в окружении, где известны переменные `even?` и `odd?`. Мы говорим «известны», потому что хотя для этих переменных и созданы привязки, их значения не имеют смысла, так как они ещё не были правильно инициализированы. Про `even?` и `odd?` известно достаточно, чтобы ссылаться на них, но пока ещё недостаточно, чтобы они участвовали в вычислениях.

Однако, такое преобразование не совсем корректно из-за порядка вычислений: действительно, `let` раскрывается в применение функции, следовательно, `letrec`, по идее, должна вести себя так же, а это значит, что начальные значения переменных должны вычисляться как аргументы функции — то есть в неопределённом порядке. К сожалению, подобный вариант всегда вычисляет их слева направо. [**см. упр. 2.9**]

Уравнения и `letrec`

С формой `letrec` есть ещё одна серьёзная проблема: её синтаксис не является строгим. При текущей трактовке `letrec` допускает в качестве инициализаторов всё что угодно, не только функции; тогда как `labels` в Common Lisp разрешает определять исключительно функции. То есть в Scheme теоретически можно будет написать следующее:

```
(letrec ((x (/ (+ x 1) 2))) x)
```

Заметьте, что переменная `x` фактически определяется через саму себя. Это, похоже, обыкновенное уравнение

$$x = \frac{x+1}{2}$$

Логично будет сделать значением `x` корень этого уравнения. То есть такое выражение должно вернуть `1`.

Но что делать, если у уравнения нет корней или если их несколько?

```
(letrec ((x (+ x 1))) x)              ; x = x + 1
(letrec ((x (+ (power x 37) 1))) x) ; x = x^37 + 1
```

Однако, существуют множества, вроде известного вам множества S-выражений, где достаточно легко убедиться в том, что уравнение имеет единственное решение [MS80]. Например, следующим образом можно без побочных эффектов определить бесконечный список — как корень данного «списочного» уравнения:

```
(letrec ((foo (cons 'bar foo))) foo)
```

Значением этого выражения может быть или лениво вычисляемый бесконечный список `(bar bar bar ...)`, как это сделано в [FW76, PJ87], так и закольцованная структура данных (менее дорогая с вычислительной точки зрения):

```
(let ((foo (cons 'bar 'wait)))
  (set-cdr! foo foo)
  foo )
```

Эффективно это одно и то же, но на самом деле нет. В общем, из-за всех этих неоднозначностей стоит ввести правило, запрещающее использовать переменную, определяемую `letrec`, для определения значения этой же переменной. В двух предыдущих примерах необходимо было знать значение `x` для того чтобы инициализировать `x`. Теперь они, очевидно, являются ошибочными. Однако мы помним, что порядок инициализации в Scheme должен быть неопределённым, а значит, некоторые конструкции, допускаемые данным правилом, могут быть ошибочными в одних реализациях, но работать в других. Рассмотрим следующий пример:

```
(letrec ((x (+ y 1))
         (y 2) )
  x )
```

Если `y` инициализируется до `x`, то всё в порядке. В противном случае возникает ошибка, потому что мы хотим увеличить значение переменной `y`, которая уже существует, но ещё не имеет значения. Некоторые компиляторы Scheme и ML анализируют выражения-инициализаторы и проводят топологическую сортировку для определения подходящего порядка инициализации. Естественно, такое решение тоже не всегда срабатывает; в частности, при взаимной зависимости[9] вроде такой:

```
(letrec ((x y) (y x)) (list x y))
```

Рассмотренные примеры напоминают о нашей дискуссии вокруг глобального окружения и семантики `define`. Там возникла похожая проблема: что делать с неинициализированными привязками и как узнать о том, что они вообще существуют.

2.6.5. Объявление неинициализированных привязок

Официально семантика Scheme считает `letrec` производной формой; то есть удобным, но отнюдь не обязательным сокращением. Соответственно, любую `letrec`-форму можно переписать с помощью примитивных форм Scheme. Чуть раньше мы попробовали это сделать, временно связывая переменные `letrec` со значением `void`. К сожалению, это тоже инициализация, так что обращения к неинициализированным переменным подобным образом отловить нельзя. Наша ситуация усугубляется тем, что ни одна из четырёх специальных форм Scheme не позволяет создавать «родные» неинициализированные привязки.

В первом приближении можно было бы решить проблему, используя некий объект `#<UFO>` [**см. стр. 33**] вместо `void`. С ним ничего нельзя сделать: ни при-

[9] Ведь `(42 42)` вполне подходит как корень данного уравнения, но почему именно `42`?

бавить к нему число, ни взять его `car`; однако, это всё же полноценный объект, так что его можно передать как аргумент в `cons`, а значит, следующая программа не будет ошибочной и вернёт `#<UFO>`:

```
(letrec ((foo (cons 'foo foo))) (cdr foo))
```

Причина такого поведения в том, что неинициализированность — это свойство самой привязки, а не её значения. Следовательно, мы не сможем решить проблему, используя объекты первого класса.

И всё же, многие реализации дают неинициализированным переменным специальное значение. Давайте назовём его `#<uninitialized>` и предположим, что это полноценный объект. Любая переменная с таким значением считается неинициализированной. Следовательно, используя вместо `void` значение `#<uninitialized>`, мы получаем желаемую возможность обнаружить ошибку. Однако, эта возможность чересчур явная: ничто не запрещает передавать `#<uninitialized>` в функцию как аргумент, а значит, больше нельзя предполагать, что все аргументы функции имеют значения. Мы будем вынуждены каждый раз проверять, действительно ли это так:

```
(define (fact n)
  (if (eq? n '#<uninitialized>)
      (wrong "Uninitialized n")
      (if (= n 0) 1
          (* n (fact (- n 1))) ) ) )
```

Делать так со всеми переменными — это слишком большая плата за `letrec`. Так что `#<uninitialized>` нельзя делать полноценным объектом, это должно быть особое внутреннее значение интерпретатора, которое нельзя использовать в программах. Для того, чтобы им можно было пользоваться безопасно, необходим специальный синтаксис.

Третий вариант решения состоит во введении специальной формы, создающей неинициализированные привязки. Например, перенесём синтаксис `let` из Common Lisp, выполняющей данное действие, в Scheme:

```
(let (переменная ...)
  ... )
```

Если имя переменной указано само по себе, без начального значения, то привязка к этому имени не будет инициализирована. Если нам понадобится её значение, то мы будем вынуждены проверять, была ли инициализирована данная переменная или нет. Теперь можно будет написать нормальную реализацию `letrec`. В следующем коде переменные $temp_i$ являются «гигиеничными»: им выдаются специальные имена, гарантированно не конфликтующие с именами переменных `letrec` или свободными переменными её тела.

```
(letrec ((имя1 выражение1)          (let (имя1 ... имяn)
         ...                          (let ((temp1 выражение1)
         (имяn выражениеn) )    ≡            ...
  тело )                                     (tempn выражениеn) )
                                       (set! имя1 temp1)
                                       ...
                                       (set! имяn tempn)
                                       тело ) )
```

Итого, проблема решена с приемлемой эффективностью: лишь неинициализированные переменные вызывают накладные расходы, потому что за особенности надо платить. Но теперь форма `let` не является просто синтаксическим сахаром, теперь это полноценная специальная форма, которую должен обрабатывать лично интерпретатор. Добавляем соответствующий код в `evaluate`:

```
...
((let)
 (eprogn (cddr e)
         (extend env
                 (map (lambda (binding)
                        (if (symbol? binding) binding
                            (car binding) ) )
                      (cadr e) )
                 (map (lambda (binding)
                        (if (symbol? binding) the-uninitialized-marker
                            (evaluate (cadr binding) env) ) )
                      (cadr e) ) ) ) ) ...
```

Переменная `the-uninitialized-marker` принадлежит языку определения. Зададим её, например, так:

```
(define the-uninitialized-marker (cons 'not 'initialized))
```

Конечно, теперь необходимо встроить поддержку этого внутреннего значения в функцию `lookup`. Функция `update!` в изменениях не нуждается по очевидным причинам. Обращения к `wrong` отвечают за два различных типа ошибок: несуществующую привязку и неинициализированную привязку.

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? (caar env) id)
          (let ((value (cdar env)))
            (if (eq? value the-uninitialized-marker)
                (wrong "Uninitialized binding" id)
                value ) )
          (lookup id (cdr env)) )
      (wrong "No such binding" id) ) )
```

После блужданий по пустыне семантики и синтаксиса, у нас наконец-то получилась форма `letrec`, позволяющая определять локальные взаимно рекурсивные функции.

2.6.6. Рекурсия без присваивания

Форма `letrec`, которую мы рассматривали, использует присваивания для обеспечения правильного вычисления начальных значений. Языки, называемые *чисто функциональными*, не имеют в своём распоряжении операторов присваивания; в них принципиально нет побочных эффектов, а чем, как не побочным эффектом вычислений, является изменение значения переменной?

В качестве парадигмы программирования запрет на присваивание имеет свои преимущества: он гарантирует сохранение ссылочной прозрачности и этим развязывает руки множеству оптимизаций, позволяя перемещать и распараллеливать части программ, использовать ленивые вычисления и т. д. Однако, если нет возможности использовать присваивания, то некоторые алгоритмы становятся не такими простыми, а также несколько усложняется перенос программ на реальные компьютеры, так как побочные эффекты являются неотъемлемой частью их работы.

Первое, что приходит в голову, это сделать `letrec` ещё одной специальной формой, как это и сделано в ML и подобных ему языках. Модифицируем `evaluate` для обработки этого случая:

```
...
((letrec)
 (let ((new-env (extend env
                        (map car (cadr e))
                        (map (lambda (binding) the-uninitialized-marker)
                             (cadr e) ) )))
      (map (lambda (binding)          ; map во имя беспорядка!
             (update! (car binding)
                      new-env
                      (evaluate (cadr binding) new-env) ) )
           (cadr e) )
      (eprogn (cddr e) new-env) ) ) ...
```

В этом случае побочные эффекты всё равно присутствуют, но на уровне интерпретатора, внутри `update!`; с точки зрения определяемого языка побочных эффектов нет. Стоит заметить, что мы намеренно не указываем порядок вычислений, используя `map`, которая, в отличие от `for-each`, вольна обрабатывать список в любом удобном порядке.[10]

[10] Правда, расплачиваясь за это необходимостью собирать бесполезный список, который тут же удаляется после создания.

`letrec` и полностью лексическое глобальное окружение

В гиперстатическом глобальном окружении переменную можно использовать только после того, как она была определена. С такими ограничениями мы не можем легко определять ни взаимно, ни даже просто рекурсивные функции. Форма `letrec` решает эту проблему, а заодно служит индикатором рекурсивных определений.

```
(letrec ((fact (lambda (n)
                 (if (= n 0) 1 (* n (fact (- n 1)))) )))
  (letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
           (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
    ... ) )
```

В данном случае `letrec` создаёт опережающие ссылки на `fact`, `odd?` и `even?`, так что определения будут работать и в гиперстатическом окружении.

Парадоксальный комбинатор

Если вы имели дело с λ-исчислением, то вы наверняка помните, что такое *комбинаторы неподвижной точки* и как записывается самый известный из них — *парадоксальный* или Y-комбинатор. Функция f имеет неподвижную точку, если в её области определения существует элемент x такой, что $f(x) = x$. Комбинатор Y принимает любую функцию λ-исчисления и возвращает её неподвижную точку. Эта идея выражена в одной из наиболее прекрасных и содержательных теорем λ-исчисления:

Теорема о неподвижной точке: $\exists \mathsf{Y} \colon \forall F \colon \mathsf{Y}F = F(\mathsf{Y}F)$

В терминах Лиспа, Y — это значение выражения

```
(let ((W (lambda (w)
           (lambda (f)
             (f ((w w) f)) ) )))
  (W W) )
```

Доказать это весьма просто. Если предположить, что Y равен (WW), то какой должна быть W, чтобы $(WW)F$ равнялось $F((WW)F)$? Очевидно, что функция W должна быть ничем иным, как $\lambda W.\lambda F.F((WW)F)$. Приведённое выражение лишь записывает эту идею на Лиспе.

Правда, здесь возникает небольшое затруднений из-за принятой в Scheme передачи аргументов по значению. Терм `((w w) f)` не следует вычислять слишком рано, поэтому мы вынуждены добавить излишнюю (в λ-исчислении) η-конверсию, чтобы избежать проблем. В итоге мы приходим к так называемому Z-комбинатору, где `(lambda (x) (... x))` означает η-конверсию:

```
(define fix
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda (x) (((w w) f) x))) ) )))
  (d d) ) )
```

Самое сложное в этом определении — понять, как оно работает. Сейчас мы этим и займёмся. Определим функцию `meta-fact`:

```
(define (meta-fact f)
  (lambda (n)
    (if (= n 0) 1
        (* n (f (- n 1))) ) ) )
```

Эта функция подозрительно похожа на факториал. Проверив, мы убеждаемся, что `(meta-fact fact)` вычисляет факториал с таким же успехом, что и `fact`, разве что несколько медленнее. Теперь предположим, что мы знаем неподвижную точку f функции `meta-fact`: $f =$ `(meta-fact` f`)`. Эта неподвижная точка по определению является решением следующего функционального уравнения относительно f:

```
f = (lambda (n)
      (if (= n 0) 1
          (* n (f (- n 1))) ) )
```

Итак, что же такое f? Не что иное, как всем известный факториал!

Вообще-то говоря, нет ни единого основания полагать, что уравнение выше имеет решение и что оно единственно. (Конечно, эти термины надо бы определить строго математически, но это выходит за рамки данной книги.) Действительно, есть как минимум ещё одно решение:

```
(define (another-fact n)
  (cond ((< n 1) (- n))
        ((= n 1) 1)
        (else (* n (another-fact (- n 1)))) ) )
```

Проверьте, пожалуйста, что `another-fact` также является неподвижной точкой `meta-fact`. Анализируя возможные неподвижные точки, можно прийти к выводу, что есть такая область определения, на которой их значения совпадают: все они вычисляют факториал натуральных чисел. Их поведение различно только тогда, когда исходный вариант `fact` попадает в бесконечный цикл. Для отрицательных целых чисел `another-fact` возвращает одно значение, хотя вполне могла бы вернуть какое-нибудь другое, потому что исходное функциональное уравнение не указывает,[11] что делать в таком случае. Если упорядочить функции по некоторой мере их определённости, то должна существовать наименьшая неподвижная точка — наименее определённое решение функционального уравнения.

[11] Более подробное объяснение см. в [Man74].

Математический смысл глобальных рекурсивных определений вроде `fact` состоит в том, что они определяют функции, являющиеся наименьшими неподвижными точками соответствующих функциональных уравнений. Когда мы пишем:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

то фактически записываем уравнение относительно переменной `fact`. Форма `define` решает это уравнение и связывает полученное решение с переменной `fact`. Такая трактовка уводит нас далеко от обсуждения инициализации глобальных переменных [**см. стр. 77**] и превращает `define` в магический решатель уравнений. В действительности, `define` реализована именно так, как предложено ранее. Просто рекурсия в глобальном окружении вместе с нормальным порядком вычислений действительно способны находить наименьшие неподвижные точки.

А теперь вернёмся к `fix`, нашему Z-комбинатору, и проследим, как же вычисляется `((fix meta-fact) 3)`. Помните, что здесь функции не имеют побочных эффектов, а значит, результаты вычислений можно свободно подставлять друг в друга, чем мы и будем пользоваться.

```
((fix meta-fact) 3)
≡ (((d d)|
         | d ≡ (lambda (w)
                 (lambda (f)
                  (f (lambda (x)
                       (((w w) f) x)) ) ) )
     meta-fact )
    3 )

≡ (((lambda (f)                  ; шаг I
       (f (lambda (x)
            (((w w) f) x)) ))|
                             | w ≡ (lambda (w)
                                     (lambda (f)
                                      (f (lambda (x)
                                           (((w w) f) x)) ) ) )
     meta-fact )
    3 )

≡ ((meta-fact (lambda (x)
                (((w w) f) x) ))|
                                | w ≡ (lambda (w)
                                        (lambda (f)
                                          (f (lambda (x)
                                               (((w w) f) x)) ) ) )
    3 )
```

```
≡ ((lambda (n)
     (if (= n 0) 1
         (* n (f (- n 1))) ) )|
                              | f ≡ (lambda (x)
                                     (((w w) meta-fact) x) )|
                                                            | w ≡
                                                                ⋮
                                     (lambda (w)            ↩
                                       (lambda (f)
                                         (f (lambda (x)
                                              (((w w) f) x) )) ) )
   3 )
≡ (* 3 (f 2))|
              | f ≡ (lambda (x)
                      (((w w) meta-fact) x) )|
                                             | w ≡
                                                 ⋮
                            (lambda (w)          ↩
                              (lambda (f)
                                (f (lambda (x)
                                     (((w w) f) x) )) )  )
≡ (* 3 (((w w) meta-fact) 2))|
                             | w ≡ (lambda (w)
                                     (lambda (f)
                                       (f (lambda (x)
                                            (((w w) f) x) )) ) )
≡ (* 3 (((lambda (f)           ; шаг II
           (f (lambda (x)
                (((w w) f) x) )) )|
                                  | w ≡ (lambda (w)
                                          (lambda (f)
                                            (f (lambda (x)
                                                 (((w w) f) x) )) ) )
         meta-fact )
        2 ) )
```

Остановимся на минутку, чтобы заметить, что на шаге II мы получили то же самое выражение, что и на шаге I. Естественно, оно появится и в третий раз:

```
(* 3 (* 2 (((lambda (f)
             (f (lambda (x)
                  (((w w) f) x) )) )|
                                    | w ≡ (lambda (w)
                                           (lambda (f)
                                             (f (lambda (x)
                                                  (((w w) f) x) )) ) )
          meta-fact )
         1 )))
```

```
≡ (* 3 (* 2 ((meta-fact (lambda (x)
                          (((w w) meta-fact) x) ))|
                                                  | w ≡
                                    (lambda (w)          ⋮
                                      (lambda (f)        ↩
                                        (f (lambda (x)
                                             (((w w) f) x)) ) ) )
               1 )))
≡ (* 3 (* 2 ((lambda (n)
                (if (= n 0) 1
                    (* n (f (- n 1))) ) )|
                                         | f → ...
               1 )))
≡ (* 3 (* 2 (if (= n 0) 1 (* n (f (- n 1))))))|
                                               | n → 1
                                               | f → ...
≡ (* 3 (* 2 1))

→ 6
```

Обратите внимание, что в процессе вычислений мы действительно используем функцию, вычисляющую факториал. Это значение выражения:

```
(lambda (x)
  (((w w) f) x) )|
                 | f ≡ meta-fact
                 | w → (lambda (w)
                         (lambda (f)
                           (f (lambda (x)
                                (((w w) f) x)) ) ) )
```

Идея состоит в том, что благодаря самоприменению мы помним, как создать заново данную функцию, и делаем это каждый раз, когда для вычислений требуется рекурсивный вызов.

Таким образом можно получить простую рекурсию без использования побочных эффектов, только с помощью `fix`, комбинатора неподвижной точки. Благодаря Y (или `fix`), `define` можно определить как решатель рекурсивных уравнений; она принимает уравнение и связывает решение с переданным именем. В итоге, если мы передадим `define` уравнение для факториала, то с `fact` будет связано следующее значение:

```
(fix (lambda (fact)
       (lambda (n)
         (if (= n 0) 1
             (* n (fact (- n 1))) ) ) ))
```

Аналогично можно решать системы уравнений, а значит, и задавать взаимно рекурсивные функции, собирая их уравнения воедино:

```
(define odd-and-even
  (fix (lambda (f)
         (lambda (which)
           (case which
             ((odd) (lambda (n) (if (= n 0) #f
                                    ((f 'even) (- n 1)) )))
             ((even) (lambda (n) (if (= n 0) #t
                                     ((f 'odd) (- n 1)) ))) ) ) )) )
(define odd? (odd-and-even 'odd))
(define even? (odd-and-even 'even))
```

У этого метода есть один большой недостаток: неэффективность, даже по сравнению с наивной реализацией `letrec`. (И всё же, см. [Roz92, Ser93].) Тем не менее, он используется, особенно в качестве книжного примера. Функциональные языки, по мнению [PJ87], тоже особо не жалуют данный метод, так как, во-первых, он неэффективен, а во-вторых, `fix` плохо сочетается с системами вывода типов. Действительно, `fix` принимает функционал,[12] принимающий функцию типа $\alpha \to \beta$, и возвращает неподвижную точку этого функционала. То есть типом `fix` является

$$\big((\alpha \to \beta) \to (\alpha \to \beta)\big) \to (\alpha \to \beta)$$

Но в определении `fix` есть самоприменение: `(d d)`. Обозначив его тип γ, имеем:

$$\gamma = \gamma \to (\alpha \to \beta)$$

Потребуется или нетривиальная система типов, чтобы в ней можно было выразить подобный рекурсивный тип, или же мы будем вынуждены реализовать `fix` в интерпретаторе как примитивную функцию, так как её нельзя выразить средствами самого языка.

2.7. Заключение

В этой главе мы прошлись по наиболее заметным из вопросов, на которые сообщество Лиспа за последние несколько десятков лет так и не смогло дать однозначного ответа. Рассмотрев причины данных разногласий, мы поняли, что они вовсе не такие серьёзные по своей сути. Большая часть из них связана с неоднозначностью толкования смысла формы `lambda` и различными способами применения функций. Хотя идея функции достаточно хорошо проработана в математике, но в функциональных (!) языках вроде Лиспа это отнюдь не так. Различные мнения по таким вопросам — это часть истории Лиспа. Подобно изучению истории родного народа, их знание облегчает понимание причин тех или иных решений в дизайне языка, а также улучшает стиль программирования в общем.

[12] Терминология Маккарти из [MAE+62]: функционал — это функция, принимающая другие функции как аргументы.

Также данная глава демонстрирует существенную важность понятия связывания. В Lisp_1 переменная (имя) ассоциируется с уникальной привязкой (возможно глобальной), которая в свою очередь ассоциируется с каким-либо значением. Так как привязка уникальна, то мы говорим о значении переменной, а не о значении привязки этой переменной. Если рассматривать привязки как абстрактный тип данных, то можно сказать, что объекты этого типа создаются связывающими формами, их значение определяется вычислением, изменяются они присваиванием, и могут быть захвачены при создании замыкания, если тело замыкания ссылается на переменную, которая ассоциирована с данной привязкой.

Привязки не являются полноценными объектами. Они не существуют в отрыве от переменных и могут быть изменены только косвенно. Собственно, привязки полезны именно потому, что они крепко-накрепко связаны со своими переменными.

Бок о бок со связывающими формами следует идея областей видимости. Область видимости переменной — это пространство в тексте программы, где можно обращаться к данной переменной. Область видимости переменных, создаваемых формой `lambda`, ограничена телом данной формы. Поэтому она называется текстуальной или лексической.

Присваивание вносит множество неоднозначностей в идею связывания, мы изучим этот вопрос подробнее в следующих главах.

2.8. Упражнения

Упражнение 2.1 Следующее выражение записано на Common Lisp. Как бы вы его перевели на Scheme?

```
(funcall (function funcall) (function funcall) (function cons) 1 2)
```

Упражнение 2.2 Что вернёт данная программа на псевдо-Common Lisp из этой главы? О чём она вам напоминает?

```
(defun test (p)
  (function bar) )

(let ((f (test #f)))
  (defun bar (x) (cdr x))
  (funcall f '(1 . 2)) )
```

Упражнение 2.3 Реализуйте в вашем интерпретаторе первые две инновации из раздела 2.3 [**см. стр. 60**]. Речь идёт о трактовке чисел и списков как функций.

Упражнение 2.4 Можно научить функцию `assoc/de` явно принимать компаратор (вроде `eq?`, `equal?` и т. п.) через аргумент, а не задавать его внутри. Сделайте это.

Упражнение 2.5 Используя `bind/de` и `assoc/de`, напишите макросы, эмулирующие специальные формы `dynamic-let`, `dynamic` и `dynamic-set!`.

Упражнение 2.6 Напишите функции `getprop` и `putprop`, которые реализуют списки свойств. Любой символ имеет личный список свойств в виде пар «ключ — значение»; добавление в этот список осуществляет функция `putprop`, поиск значения по ключу осуществляет функция `getprop`. Также, естественно, должно выполняться утверждение

```
(begin (putprop 'symbol 'key 'value)
       (getprop 'symbol 'key) )      → value
```

Упражнение 2.7 Определите специальную форму `label` на Lisp_1.

Упражнение 2.8 Определите специальную форму `labels` на Lisp_2.

Упражнение 2.9 Придумайте, как реализовать `letrec` с помощью `let` и `set!` так, чтобы порядок вычисления значений-инициализаторов был неопределённым.

Упражнение 2.10 У нашего комбинатора неподвижной точки на Scheme обнаружился недостаток: он поддерживает только унарные функции. Реализуйте `fix2`, работающий с бинарными функциями. Затем `fixN`, поддерживающий функции любой арности.

Упражнение 2.11 Далее напишите функцию `NfixN`, возвращающую неподвижные точки для списка функционалов произвольной арности. Её можно использовать, например, следующим образом:

```
(let ((odd-and-even
       (NfixN (list (lambda (odd? even?)    ; odd?
                      (lambda (n)
                        (if (= n 0) #f (even? (- n 1))) ) )
                    (lambda (odd? even?)    ; even?
                      (lambda (n)
                        (if (= n 0) #t (odd? (- n 1))) ) ) )) ))
  (set! odd? (car odd-and-even))
  (set! even? (cadr odd-and-even)) )
```

Упражнение 2.12 Рассмотрим функцию `klop`. Является ли она комбинатором неподвижной точки? Попробуйте доказать или опровергнуть, что `(klop` f`)` тоже возвращает неподвижную точку f подобно `fix`.

```
(define klop
  (let ((r (lambda (s c h e m)
             (lambda (f)
               (f (lambda (n)
                    (((m e c h e s) f) n)) ) ) )))
    (r r r r r r) ) )
```

Упражнение 2.13 Если функция `hyper-fact` определена так:

```
(define (hyper-fact f)
  (lambda (n)
    (if (= n 0) 1
        (* n ((f f) (- n 1))) ) ) )
```

то что вернёт `((hyper-fact hyper-fact) 5)`?

Рекомендуемая литература

Кроме упомянутой ранее работы по λ-исчислению [SS78a] также имеет смысл почитать про анализ функций в [Mos70] и сравнительный анализ Lisp_1 и Lisp_2 в [GP88].

В [Gor88] есть интересное введение в λ-исчисление.

Комбинатор Y разбирается подробнее в [Gab88].

Глава 3

Переходы и возвраты: продолжения

Каждое вычисление в конечном счёте приводит к возврату результата сущности, которая называется *продолжением*. В данной главе разбирается эта идея и её исторические предпосылки. Мы также создадим ещё один интерпретатор, призванный явно оперировать продолжениями. В процессе разработки будут рассмотрены различные варианты реализации продолжений в Лиспе и Scheme, а также своеобразный «стиль передачи продолжений». Одним из отличий Лиспа от других языков является большое количество механизмов управления ходом вычислений. Это в некотором смысле превращает данную главу в каталог [Moz87], где представлена тысяча и одна управляющая конструкция. С другой стороны, мы не будем вдаваться в подробности о продолжениях; по крайней мере, о том, как физически реализуется их захват и сохранение. Наш интерпретатор будет использовать объекты для представления продолжений в виде *стека вызовов*.

Интерпретаторам, построенным нами ранее, было необходимо только окружение, чтобы вычислить значение переданного выражения. К сожалению, они не в состоянии проводить вычисления, в которых есть *переходы* (escapes): полезная управляющая конструкция, позволяющая покинуть текущий контекст исполнения, чтобы перейти в другой, более подходящий. Обычно они используются для обработки исключительных ситуаций, когда мы указываем, куда нам следует перейти для обработки события или ошибки, прервавшей нормальный ход вычислений.

История переходов в Лиспе восходит ко временам Lisp 1.5 и формы `prog`. Сейчас эта форма считается устаревшей, но раньше на неё возлагались большие надежды по переманиванию программистов на Алголе в ряды лисперов, так как считалось, что для них было более привычным использование `goto`. Вместо этого оказалось, что данная форма больше влияет на самих лисперов, сталкивая их с праведного пути хвостовой рекурсии.[1] Тем не менее, форма `prog` достойна рассмотрения, потому как обладает несколькими интересными свойствами. Например, вот так с её помощью записывается факториал:

[1] Например, сравните стили изложения первого и третьего изданий [WH89].

Common Lisp
```
(defun fact (n)
  (prog (r)
            (setq r 1)
       loop (cond ((= n 1) (return r)))
            (setq r (* n r))
            (setq n (- n 1))
            (go loop) ) )
```

Специальная форма `prog` сначала объявляет все используемые локальные переменные (в данном случае это `r`). Далее следуют инструкции (представляемые списками) и метки (представляемые символами). Инструкции последовательно вычисляются, как в `progn`. Результатом вычисления формы `prog` по умолчанию является `nil`. Но внутри `prog` можно использовать специальные инструкции. Безусловные переходы выполняются с помощью `go` (которая принимает символ — имя метки), а вернуть определённое значение из `prog` можно с помощью `return`. В Lisp 1.5 было лишь одно ограничение: формы `go` и `return` могли появляться только на первом уровне вложенности или внутри `cond` на том же первом уровне.

Форма `return` позволяла выйти из `prog`, забрав с собой результат вычислений. Ограничение Lisp 1.5 допускало лишь простые переходы, в более поздних версиях оно было снято, что позволило реализовать более изощрённые варианты поведения. Переходы стали обычным способом обработки ошибок. Если происходила ошибка, то выполнение переходило из ошибочного контекста исполнения в безопасный для обработки возникшей ситуации. Теперь можно переписать факториал следующим образом, поместив `return` глубже:

Common Lisp
```
(defun fact2 (n)
  (prog (r)
            (setq r 1)
       loop (setq r (* (cond ((= n 1) (return r))
                             ('else n) )
                       r ))
            (setq n (- n 1))
            (go loop) ) )
```

Если рассматривать формы `prog` и `return` только как управляющие конструкции, то становится ясно, что они влияют на последовательность вычислений подобно функциям: выполнение функции начинается переходом в её тело и заканчивается возвратом результата в то место, откуда функция была вызвана. Только в нашем случае внутри формы `prog` известно, куда требуется вернуть значение, — она сама связывает `return` с этим местом. Для перехода не требуется знать, откуда мы уходим, но необходимо знать, куда мы хотим попасть.

Если такой *прыжок* будет эффективно реализован, то это порождает жизнеспособную парадигму программирования. Например, пусть стоит задача проверить вхождение элемента в двоичное дерево. В лоб эта задача решается примерно таким способом:

```
(define (find-symbol id tree)
  (if (pair? tree)
      (or (find-symbol id (car tree))
          (find-symbol id (cdr tree)) )
      (eq? tree id) ) )
```

Допустим, мы ищем `foo` в следующем дереве: `(((a . b) . (foo . c)) . (d . e))`. Так как поиск идёт слева направо и в глубину, то после того, как нужный символ будет найден, нам ещё предстоит подниматься обратно по вложенным `or`, неся с собой вожделенную `#t`, которая в конце концов станет результатом вычислений. Вот так это происходит:

```
(find-symbol 'foo '(((a . b) . (foo . c)) . (d . e)))
≡ (or (find-symbol 'foo '((a . b) . (foo . c)))
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo '(a . b))
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or (or (find-symbol 'foo 'a)
              (find-symbol 'foo 'b) )
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo 'b)
          (find-symbol 'foo '(foo . c)) )
      (find-symbol 'foo '(d . e)) )
≡ (or (find-symbol 'foo '(foo . c))
      (find-symbol 'foo '(d . e)) )
≡ (or (or (find-symbol 'foo 'foo)
          (find-symbol 'foo 'c) )
      (find-symbol 'foo '(d . e)) )
≡ (or (or #t
          (find-symbol 'foo 'c) )
      (find-symbol 'foo '(d . e)) )
≡ (or #t
      (find-symbol 'foo '(d . e)) )
→ #t
```

Как раз здесь бы не помешал эффективно реализованный переход к последней строке. Как только мы находим нужный символ, то не продираемся через `or` и уж тем более не смотрим в другие ветки, а сразу же возвращаем результат.

Другим примером может быть так называемое программирование исключениями. Суть подхода: пусть в цикле выполняются какие-то действия. Данный цикл продолжает выполняться до тех пор, пока не возникает исключительная ситуация и не происходит выход из цикла, который иначе бы продолжался вечно. Нечто подобное реализует функция `better-map`, рассматриваемая позже. [**см. стр. 106**]

Размышляя дальше над природой сущности, представляющей точку входа в функцию, можно прийти к выводу, что понятие вычислений подразумевает не только выражение, которое необходимо вычислить, и окружение, в котором будут проходить вычисления, но и нечто, куда необходимо вернуть полученный результат. Это нечто и называется *продолжением* (continuation). Это всё, что ещё осталось вычислить.

У любого вычисления есть продолжение. Например, в выражении `(+ 3 (* 2 4))` продолжением подвыражения `(* 2 4)` будет сложение, где первый аргумент это `3`, а второй ожидается в результате вычислений. Здесь можно заметить параллели и представить продолжения в более привычной форме — как функции. Ведь продолжения тоже представляют некоторые вычисления и, как и функции, тоже требуют, чтобы сначала были вычислены все необходимые параметры. Для предыдущего примера продолжением `(* 2 4)` будет функция `(lambda (`x`) (+ 3 `x`))`, подчёркивающая тот факт, что вычисление ожидает второй аргумент для сложения.

Продолжения можно записывать и проще, в духе λ-исчисления. Мы будем записывать предыдущее продолжение как `(+ 3 [])`, где `[]` означает место, куда необходимо подставить результат вычислений.

Действительно, у всего есть продолжение. Вычисление условного выражения в формах ветвления проводится для продолжения, которое ожидает это значение, чтобы выбрать ту или иную ветку условной формы. В выражении `(if (foo) 1 2)` продолжением вызова `(foo)` является `(lambda (`x`) (if `x` 1 2))` или `(if [] 1 2)`.

Переходы, исключения и тому подобные механизмы — это лишь частные случаи манипуляции продолжениями. Имея это в виду, давайте рассмотрим в деталях различные варианты использования продолжений, которые были придуманы за последние тридцать с лишним лет.

3.1. Формы, манипулирующие продолжениями

Явное использование продолжений даёт нам возможность управлять ходом исполнения программы. Форма `prog` имеет схожие возможности, но при этом обладает излишним функционалом `let`. Оставив только функционал управления потоком исполнения, мы получим то, для чего (в первую очередь) были придуманы формы `catch` и `throw`.

3.1.1. Пара `catch`/`throw`

Специальная форма `catch` имеет следующий синтаксис:

`(catch` *метка формы*`...)`

Метка вычисляется и с ней связывается продолжение формы `catch`. Раз связывается, то это значит, что нам необходимо новое пространство имён —

динамическое окружение меток, — в котором и будут храниться эти связи. Это не совсем пространство *имён*, так как метки не обязательно являются идентификаторами, но вполне на него похоже по смыслу. Правда, произвольность значений меток может вызвать проблемы с определением их равенства и, следовательно, с поиском в этом окружении, так как не все значения можно легко и однозначно сравнивать. К этому вопросу мы ещё вернёмся.

Оставшиеся *формы* являются телом `catch` и вычисляются последовательно, как в `progn` или `begin`. Если ничего не произошло, то значением `catch` является значение последней вычисленной формы. Но мы можем вмешаться в это поведение с помощью `throw`.

Форма `throw` имеет следующий синтаксис:

`(throw` *метка форма*`)`

Первый аргумент должен вычисляться в значение, с которым `catch` динамически связала продолжение. Если это так, то исполнение переходит в соответствующее продолжение, а вместо значения `catch` подставляется значение *формы* из `throw`.

Вернёмся к примеру с поиском в двоичном дереве и перепишем его с использованием `catch` и `throw`. Мы не будем здесь бессмысленно передавать по рекурсивным вызовам значение `id`, так как оно лексически видимо отовсюду; такое поведение реализовано с помощью вспомогательной функции.

```
(define (find-symbol id tree)
  (define (find tree)
    (if (pair? tree)
        (or (find (car tree))
            (find (cdr tree)) )
        (if (eq? tree id)
            (throw 'find #t)
            #f ) ) )
  (catch 'find
    (find tree) ) )
```

Форма `catch`, оправдывая своё название, ловит значение, которое бросает ей `throw`. Переход в данном случае обеспечивается явным указанием значения, связанного с сохранённым продолжением. То есть `catch` — это связывающая форма, которая ассоциирует метку с текущим продолжением. Тогда форма `throw` фактически ссылается на это продолжение, используя его для управления потоком вычислений. Сама по себе она не возвращает значения, `throw` лишь заставляет `catch` вернуть указанное значение. Здесь `catch` захватывает продолжение вызова `find-symbol`, а `throw` выполняет прямой переход к дальнейшим вычислениям, которые должны выполняться после вызова `find-symbol`.

Динамическое окружение меток описывается следующей таблицей свойств:

Ссылка	`(throw` *метка* `...)`
Значение	отсутствует, это объекты второго класса
Изменение	запрещено
Расширение	`(catch` *метка* `...)`
Определение	запрещено

Как мы уже говорили, `catch` это не функция, а специальная форма, вычисляющая свой первый аргумент (метку), затем связывающая с ней в динамическом окружении своё продолжение, после чего вычисляющая оставшиеся формы подобно `begin`. Не обязательно все они будут вычислены. Когда `catch` возвращает значение или мы выходим из неё с помощью `throw`, связь между меткой и продолжением автоматически удаляется.

Форму `throw` же можно реализовать и как функцию, и как специальную форму. Если это специальная форма, как в Common Lisp, то она вычисляет метку, затем ищет соответствующее продолжение `catch`, и, если находит, то вычисляет значение для передачи и выполняет переход. Если же `throw` реализована как функция, то всё происходит немного в другом порядке: сначала вычисляются оба аргумента, затем ищется `catch`, после чего выполняется переход.

Эти семантические различия хорошо показывают неточность описания поведения этих форм естественным языком. И на этом они не заканчиваются, можно придумать ещё множество вопросов, на которые будет сложно дать однозначный ответ. Например, что делать, если соответствующей `catch`-формы нет? Как именно всё же сравнивать метки? Что будет, если написать `(throw` α `(throw` β π`))`? Мы попытаемся ответить на эти вопросы немного позже.

3.1.2. Пара `block`/`return-from`

Переходы, которые реализуют `catch` и `throw`, выполняются динамически. Когда `throw` запрашивает переход, она должна во время исполнения программы отыскать соответствующую `catch`-форму и её продолжение. Естественно, это требует времени, которое можно попытаться сократить, использовав *лексические* метки, как их называют в Common Lisp. Специальные формы `block` и `return-from` слегка напоминают `catch` и `throw`.

Форма `block` имеет следующий синтаксис:

`(block` *метка формы*`...)`

Первый аргумент не вычисляется и должен быть идентификатором. Форма `block` связывает текущее продолжение с *меткой* в *лексическом окружении меток*. Далее вычисляется тело `block` как в `progn` и последнее полученное значение становится значением всей формы `block`. Последовательность можно прервать с помощью `return-from`.

Форма `return-from` имеет следующий синтаксис:

`(return-from` *метка форма*`)`

Первый аргумент не вычисляется и должен быть именем лексически видимой метки; читай: `return-from` может находиться только внутри `block` с одноимённой меткой, как переменная может использоваться только внутри соответствующей `lambda`-формы. При вычислении `return-from` соответствующий `block` прерывается и возвращает значение *формы*.

Лексические метки образуют новое пространство имён, чьи свойства описываются следующей таблицей:

Ссылка	`(return-from` *метка* `...)`
Значение	отсутствует, это объекты второго класса
Изменение	запрещено
Расширение	`(block` *метка* `...)`
Определение	запрещено

Перепишем[2] наш пример с деревом на новый лад:

```
(define (find-symbol id tree)
  (block find
    (letrec ((find (lambda (tree)
                     (if (pair? tree)
                         (or (find (car tree))
                             (find (cdr tree)) )
                         (if (eq? id tree)
                             (return-from find #t)
                             #f ) ) )))
      (find tree) ) ) )
```

Заметьте, мы не просто поменяли все „`catch 'find`“ на „`block find`“; нам потребовалось переместить тело функции внутрь `block`, иначе `return-from` не находилась бы в лексической области видимости метки `find`.

Компилятор может генерировать для `block` очень эффективный код. Грубо говоря, всё, что надо сделать `block`, — это сохранить высоту стека вызовов в соответствующей метке. А `return-from` надо лишь положить возвращаемое значение `#t` туда, где его ждут (в регистр, например), после чего вернуть указатель верхушки стека в положение, сохранённое в метке `find`. Это всего лишь пара-тройка инструкций, а не поиск среди всех доступных меток, который устраивает `catch`. Отличие станет более заметным, когда мы попробуем реализовать `catch`[3] с помощью `block`:

[2] Нам пришлось воспользоваться `letrec`, так как в Scheme запрещено располагать `define` сразу внутри `block`, а `(let () (define ...))` писать тоже нельзя из-за `()`.

[3] Этот вариант `catch` использует `block` с меткой `label`. Естественно, система макросов должна обеспечивать гигиеничность имён (гарантировать отсутствие коллизий) и для окружения меток.

```
(define *active-catchers* '())

(define-syntax throw
  (syntax-rules ()
    ((throw tag value)
     (let* ((label tag)   ; вычисляется единожды
            (escape (assv label *active-catchers*)) ) ; узкое место
       (if (pair? escape)
           ((cdr escape) value)
           (wrong "No associated catch to" label) ) ) ) ) )

(define-syntax catch
  (syntax-rules ()
    ((catch tag . body)
     (let* ((saved-catchers *active-catchers*)
            (result (block label
                      (set! *active-catchers*
                            (cons (cons tag
                                        (lambda (x)
                                          (return-from label x) ) )
                                  *active-catchers* ) )
                      . body )) )
       (set! *active-catchers* saved-catchers)
       result ) ) ) )
```

Здесь практически вся стоимость использования `catch`/`throw` сосредоточена в вызове `assv`[4] при раскрытии макроса `throw`. Рассмотрим, как работает эта реализация. Глобальная переменная (тут она названа `*active-catchers*`) хранит все активные `catch`-формы (выполнение которых ещё не завершилось). Переменная обновляется при выходе из `catch` (как нормальном, так и при помощи `throw`). Значением `*active-catchers*` является А-список пар «метка — продолжение». Этот список фактически соответствует динамическому окружению, которым пользовались исходные `catch` и `throw` для обмена информацией о метках.

3.1.3. Метки с динамическим временем жизни

Однако, эта эмуляция не совсем хороша, так как если `catch` окажется внутри `block`, то значение переменной `*active-catchers*` может быть искажено. Эмулировать неестественный для языка синтаксис непросто [Fel90, Bak92c], так как это часто требует сложных архитектурных решений, которым необходимы ресурсы в единоличное пользование (вроде `*active-catchers*`). Позже

[4] Кстати, в этом случае равенство меток устанавливается предикатом `eqv?`.

мы покажем, как подружить `catch` и `block`, но уже с помощью специального приспособления: `unwind-protect`.[5]

Как и все объекты Лиспа, продолжения тоже имеют своё время жизни. В эмуляции `catch` с помощью `block` видно, что продолжение, захватываемое `catch`, живёт только во время вычислений внутри тела `catch`. Это называется *динамическим* временем жизни. Такое поведение напоминает динамические переменные, которые тоже существуют только во время вычислений внутри связывающей формы, что их создала. Поэтому давайте сделаем список `*active-catchers*` динамической переменной, что позволит использовать `catch` и `block` одновременно, так как задача поддержания целостности списка будет передана механизму динамических переменных.

```
(define-syntax throw
  (syntax-rules ()
    ((throw tag value)
     (let* ((label tag)
            (escape (assv label (dynamic *active-catchers*))) )
       (if (pair? escape)
           ((cdr escape) value)
           (wrong "No associated catch to" label) ) ) ) ) )

(define-syntax catch
  (syntax-rules ()
    ((catch tag . body)
     (block label
       (dynamic-let ((*active-catchers*
                      (cons (cons tag (lambda (x)
                                        (return-from label x) ))
                            (dynamic *active-catchers*) ) ))
         . body ) ) ) ) )
```

Время жизни метки, создаваемой `block` в Common Lisp, динамическое, так что на метку можно перейти только внутри тела `block`. Точно такая же ситуация и с `catch`. Однако, лексический характер привязки, создаваемой `block`, является источником проблемы, которая не возникает с `catch`: если `throw` и `return-from` позволяют отбросить оставшиеся вычисления, то эти вычисления должны существовать в момент совершения перехода. Рассмотрим следующую программу:

```
((block foo
   (lambda (x) (return-from foo x)) )
 33 )
```

[5] Альтернативное решение: эмулировать `block` и `return-from` с помощью самих себя таким образом, чтобы они учитывали `catch` и `throw`. Это сложновато реализовать прямиком в интерпретаторе, но вполне возможно.

Здесь функция-переход на `foo` применяется к числу 33, но ведь в момент применения этой функции уже нет никакой метки `foo`, так что мы получаем ошибку. При создании замыкания оно честно захватило своё окружение, в частности, метку `foo`. Потом это замыкание возвращается как значение `block` и мы выходим из этой формы. Нельзя выйти оттуда ещё раз, перейдя на `foo`, так как вычисления уже завершены. Поэтому во время вызова подобных функций необходимо проверять, не стало ли сохранённое продолжение уже неактуальным, и выполнять переход только при соблюдении этого условия. Кроме того, не стоит забывать и о другом аспекте лексических меток, создаваемых `block`. Например:

```
(block foo
  (let ((f1 (lambda (x) (return-from foo x))))
    (* 2 (block foo
           (f1 1)) ) ) → 1
```

Сравните это с результатом, который мы получим, сменив „`block foo`“ на „`catch 'foo`“:

```
(catch 'foo
  (let ((f1 (lambda (x) (throw 'foo x))))
    (* 2 (catch 'foo
           (f1 1)) ) ) → 2
```

Функция `f1` в данном случае вызовет выход из ближайшей формы `catch`, ожидающей `foo`, а не той, которая была видна ей при определении; соответственно, умножение будет выполнено и верхняя форма `catch` получит значение 2.

3.1.4. Сравнение `catch` и `block`

С одной стороны, `catch` и `block` похожи: захватываемые ими продолжения имеют динамическое время жизни — ими можно пользоваться только внутри соответствующих блоков. С другой стороны, `return-from` всегда обращается к нужному продолжению, тогда как `throw` может их перепутать. Форма `block` более эффективна, так как `return-from` не требуется проверять, действительно ли существует парный ему `block`, — это гарантируется синтаксисом. Тем не менее, ей приходится проверять, можно ли воспользоваться сохранённым продолжением, хотя чаще всего это тоже можно гарантировать по исходному коду. Видна явная параллель между динамическими и лексическими метками с одной стороны и динамическими и лексическими переменными с другой: возникающие проблемы схожи в обоих случаях.

Динамические метки могут конфликтовать между собой — лексические же в принципе не могут мешать друг другу. Взять хотя бы возможность использовать динамические метки где угодно: одна функция может случайно перехватить то, что предназначалось другой. Например:

```
(define (foo)
  (catch 'foo (* 2 (bar))) )

(define (bar)
  (+ 1 (throw 'foo 5)) )

(foo) → 5
```

`block` ограничивает область видимости метки лишь своим телом, тогда как `catch` отзывается на подходящий `throw` из любого уголка программы. Поэтому можно будет сделать `(throw 'foo ...)` где угодно в процессе вычисления `(* 2 (bar))` и это сработает. Использование „`catch 'foo`“ вместо „`block foo`“ естественно приведёт к ошибке, так как `return-from` понятия не имеет о `(block foo ...)` в функциях, которые вызывают `bar`. Но это ещё безобидный пример. Рассмотрим следующую ситуацию.

```
(catch 'not-a-pair
  (better-map (lambda (x)
                (or (pair? x)
                    (throw 'not-a-pair x) ) )
              (hack-and-return-list) ) )
```

Предположим, мы услышали от бабушек на лавочке, что `better-map` гораздо лучше `map`; далее допустим, что мы рискнём использовать именно её, чтобы быстро проверить, действительно ли `(hack-and-return-list)` возвращает список, состоящий из пар; наконец предположим, что мы не знаем, как реализована `better-map`, хотя на самом деле она выглядит вот так:

```
(define (better-map f L)
  (define (loop L1 L2 flag)
    (if (pair? L1)
        (if (eq? L1 L2)
            (throw 'not-a-pair L)
            (cons (F (car L1))
                  (loop (cdr L1)
                        (if flag (cdr L2) L2)
                        (not flag) ) ) ) ) )
  (loop L (cons 'ignore L) #t) )
```

Функция `better-map` интересна тем, что в отличие от `map` она не зависает намертво на замкнутых списках (при правильном использовании). Если `(hack-and-return-list)` возвращает следующий список: `#1=((foo . hack) . #1#)`,[6] то `better-map` прыгнет на ближайшую метку `not-a-pair` и таким образом избежит бесконечного цикла. Но вот незадача: замыканию, переданному в `better-map`, эта метка тоже нужна. Конечно, если в документации

[6] Здесь используется нотация Common Lisp для рекурсивных структур данных. Такой список в Scheme возвращает выражение `(let ((p (list (cons 'foo 'hack)))) (set-cdr! p p) p)`.

к `better-map` будет написано, что она использует такую метку, то конфликт имён можно устранить, использовав какое-нибудь другое имя для своих целей. Благо, в случае `catch` можно использовать что угодно в качестве метки, в частности, список, собранный специально для этого случая:

```
(let ((tag (list 'not-a-pair)))
  (catch tag
    (better-map (lambda (x)
                  (or (pair? x)
                      (throw tag x) ) )
                (hack-and-return-list) ) ) )
```

Наконец, рассмотрим, как можно проэмулировать `block` с помощью `catch` (естественно, выигрыша в производительности мы не получим). Для этого достаточно обеспечить лексичность используемых меток.

```
(define-syntax block
  (syntax-rules ()
    ((block label . body)
     (let ((label (list 'label)))
       (catch label . body) ) ) ) )
(define-syntax return-from
  (syntax-rules ()
    ((return-from label value)
     (throw label value) ) ) )
```

Макрос `block` создаёт уникальную метку и лексически связывает её с одноимённой переменной. Этим мы гарантируем, что нужную метку будут видеть только те `return-from`, которые лексически находятся внутри формы `block`. Правда, для этого используется имя `label`, что опять может вызвать конфликты. Конечно, можно воспользоваться чем-то вроде `gensym`, достаточно лишь удостовериться, что `catch` и `throw` используют одинаковые метки.

3.1.5. Метки с неограниченным временем жизни

Диалект Scheme, появившийся около 1975 года, предложил дать продолжениям, захватываемым `catch` и `block`, неограниченное время жизни. Это свойство открыло поразительные возможности их использования. Позже, в соответствии с догматом о минимальном количестве специальных форм, были предприняты попытки выразить захват продолжений и сами продолжения как функции. В [Lan65] Питер Лэндин предложил оператор *J*, прямым потомком которого является функция `call/cc` в Scheme.

Мы попробуем объяснить её синтаксис настолько просто, насколько это возможно. Во-первых, она захватывает продолжения, так что это должна быть форма, где доступно продолжение её вызова:

$_k$`(...)`

Далее, это должна быть функция. Назовём её `call/cc`:

$_k$`(call/cc ...)`

Теперь, когда мы захватили k, его нужно как-то передать пользователю. Но как? Очевидно, нельзя вернуть k как значение `call/cc`, потому что это бессмысленно. Оно ожидается внутри какого-то вычисления, так что можно это вычисление обернуть в унарную функцию,[7] принимающую k, которую и передать внутрь `call/cc`:

$_k$`(call/cc (lambda (k) ...))`

Продолжение k становится объектом первого класса, к которому применяется функция-аргумент `call/cc`. Аналогично, само продолжение k тоже является унарной функцией, которая неотличима от замыканий, создаваемых `lambda`. Функция `call/cc` *реифицирует* продолжение k в полноценный объект, который становится значением переменной `k`. Достаточно вызвать функцию `k`, чтобы передать её аргумент форме, вызвавшей `call/cc`:

$_k$`(call/cc (lambda (k) (+ 1 (k 2))))` $\rightarrow$ `2`

Можно было бы создать особый объект «продолжение». Тогда их уже не получится вызывать как функции, для этого необходимо специальное средство передачи управления: функция `continue`. Пример выше записывался бы тогда так:

$_k$`(call/cc (lambda (k) (+ 1 (continue k 2))))` $\rightarrow$ `2`

Тем не менее, даже в этом случае можно легко сделать продолжение функцией, обернув его в `(lambda (v) (continue` k `v))`. Некоторым людям нравится использовать развёрнутую форму, так как она делает переход более заметным.

Вот и всё. Осталось только самое сложное — запомнить полное имя этой функции: `call-with-current-continuation`. А теперь давайте перепишем наш пример с двоичным деревом, используя `call/cc`:

```
(define (find-symbol id tree)
  (call/cc
   (lambda (exit)
     (define (find tree)
       (if (pair? tree)
           (or (find (car tree))
               (find (cdr tree)) )
           (if (eq? tree id) (exit #t) #f) ) )
     (find tree) ) ) )
```

Продолжение вызова функции `find-symbol` захватывается и превращается в унарную функцию, связываемую с переменной `exit`. Как только мы находим нужный символ, поиск прерывается вызовом `exit`, после чего мы никогда

[7] В Scheme достаточно, чтобы функция могла принять как минимум один аргумент. То есть с `(call/cc list)` никаких проблем нет.

уже не возвращаемся внутрь `find-symbol`, так как дальше продолжаются вычисления, следующие после вызова `find-symbol`.

В этом примере не очевидна неограниченность времени жизни продолжения, потому что оно используется исключительно внутри самой же формы `call/cc`. Но теперь мы можем сохранить продолжение в любой переменной, чего нельзя сделать с меткой `block`/`catch`.

```
(define (fact n)
  (let ((r 1) (k 'void))
    (call/cc (lambda (c) (set! k c) 'void))
    (set! r (* r n))
    (set! n (- n 1))
    (if (= n 1) r (k 'recurse)) ) )
```

Продолжение, которое передаётся через `c` и сохраняется в `k`, выглядит так:

```
k = (lambda (u)
      (set! r (* r n))
      (set! n (- n 1))
      (if (= n 1) r (k 'recurse)) )
                                    | r → 1
                                    | k → k
                                    | n
```

Это же продолжение k связано с переменной `k` внутри самого себя. Рекурсия, как мы знаем, всегда означает какой-то цикл; в данном случае `k` вызывается до тех пор, пока `n` не достигнет желаемого значения. Всё это вместе, естественно, вычисляет факториал.

В этом примере продолжение k используется вне создавшей его формы `call/cc`. Кстати, можно избавиться от избыточных переменных и аргументов, просто возвращая это продолжение:

```
(define (fact n)
  (let ((r 1) (k (call/cc (lambda (c) c))))
    (set! r (* r n))
    (set! n (- n 1))
    (if (= n 1) r (k k)) ) )
```

Самоприменение `(k k)` необходимо, так как нам необходимо поддерживать правильное значение `k`. Это продолжение можно записать так:

```
(lambda (u)
  (let ((k u))
    (set! r (* r n))
    (set! n (- n 1))
    (if (= n 1) r (k k)) ) )
                              | r → 1
                              | n
```

Неограниченность времени жизни усложняет реализацию продолжений и в общем случае увеличивает стоимость их использования. (См. [CHO88,

HDB90, Mat92].) Почему? Потому, что в таком случае вложенные вычисления уже нельзя представлять в виде стека, здесь требуется дерево. Если продолжения имеют исключительно динамическое время жизни, то это просто переходы: с их помощью можно покинуть текущие вычисления, но только один раз. В этом случае легко понять, когда вычисление формы начинается и заканчивается: начинается при входе в неё, а заканчивается с последним выражением или первым встреченным переходом.

Если же продолжения живут неограниченно долго, то всё гораздо усложняется. Вспомните форму `(call/cc ...)` в примере с факториалом: она фактически возвращает результат несколько раз. Если допускать такую возможность (многократного возврата значений[8]), то уже не получится считать, что выполнение функции окончено, когда она вернула значение.

`call/cc` могущественна и в некотором смысле может манипулировать временем. Программа прожила какой-то промежуток времени, наделала ошибок и решила взять вторую попытку, перепрыгнув назад в прошлое (при этом она заранее предусмотрела такой исход и оставила метку в нужном ей моменте времени). При этом она забирает с собой весь прожитый опыт (свою память), так что вычисления после прыжка пойдут уже другим путём. Естественно, ничто не запрещает ей сделать и третью попытку или использовать эту силу не только для исправления ошибок, но и для собственной выгоды.

С другой стороны, форма `call/cc` очень похожа на оператор `goto`, который considered harmful. Однако `call/cc` более ограничена, так как позволяет лишь *вернуться* в те места программы, где мы уже были, но *не отправиться* туда, где нас никогда не было.

Вначале бывает нелегко научиться пользоваться `call/cc`, так как и её аргумент, и продолжение являются унарными функциями. Возможно, в таком случае вам поможет понимание `call/cc` следующим образом:

$$_k\texttt{(call/cc } \varphi\texttt{)} \to {}_k(\varphi\ k)$$

где k является продолжением вызова `call/cc`, а φ — какой-то унарной функцией. Вызов `call/cc` лишь превращает k в объект языка, который можно передать как аргумент. Заметьте, что продолжением вызова φ является всё так же k, поэтому для того, чтобы просто вернуть результат, не обязательно пользоваться переданным продолжением:

```
(call/cc (lambda (k) 1515)) → 1515
```

Кого-то такое умолчательное поведение может расстраивать. В некоторых языках `call/cc` изымает захватываемое продолжение k из хода вычислений: k больше не является её собственным продолжением, и для возврата значения последующим вычислениям его надо обязательно явно передать правильному продолжению:

```
(call/cc (lambda (k) (k 1615))) → 1615
```

[8] Здесь ударение стоит на слове «*возврат*», а не «значений». Функция `values` не имеет никакого отношения к `call/cc`.

Если этого не сделать, то продолжением формы `call/cc` будет нечто, подобное чёрной дыре: $\lambda u.\bullet$. Оно поглощает все вычисления вместе с передаваемым значением. Ничто и никогда не возвращается назад, попав в чёрную дыру. Исполнение программы состоит из вызовов функций, а они похожи на дыхание: вдох-выдох — вход-выход. Необходимым условием жизни является *продолжение* дыхания. Используя `call/cc`, мы вмешиваемся в нормальный ход исполнения, так что без определённых предосторожностей можно всерьёз забыть, как дышать, и умереть.

3.1.6. Защитные формы

Осталось рассмотреть ещё один эффект, относящийся к продолжениям. Связан он со специальной формой `unwind-protect`. Названием она обязана принципу первой реализации[9] и задуманному функциональному назначению. Вот синтаксис этой формы:

`(unwind-protect` *форма*
  формы-уборщики`...)`

Сначала вычисляется *форма*, её значение станет значением всей формы `unwind-protect`. Как только значение *формы* получено, вычисляются *формы-уборщики*, и только потом `unwind-protect` возвращает ранее вычисленное значение. Она похожа на `prog1` из Common Lisp или `begin0` из некоторых версий Scheme, которые последовательно вычисляют формы подобно `begin`, но возвращают значение первой из них, а не последней. Вот только `unwind-protect` гарантирует выполнение *уборщиков* даже в случае, если вычисление *формы* было прервано переходом. Поэтому:

Common Lisp

```
(let ((a 'on))
  (cons (unwind-protect (list a)
          (setq a 'off) )
        a ) ) → ((on) . off)

(block foo
  (unwind-protect (return-from foo 1)
    (print 2) ) ) → 1 ; и печатает 2
```

Данная форма полезна, когда состояние системы должно быть восстановлено вне зависимости от результата производимых действий. Например, когда мы читаем файл, то в конце он должен быть закрыт в любом случае. Другим примером является эмуляция `catch` с помощью `block`. Если данные формы используются одновременно, то возможна рассинхронизация состояния `*active-`

[9] Тут та же ситуация, что и с `car` и `cdr`, которые являются акронимами от «contents of the address register» и «contents of the decrement register» — с помощью данных примитивов обрабатывались точечные пары в оригинальной реализации Лиспа для IBM 704. Это не имеет ничего общего с текущими реализациями, но названия приклеились.

`catchers*`. Этот недостаток можно исправить с помощью `unwind-protect`, гарантируя восстановление `*active-catchers*`:

```
(define-syntax catch
  (syntax-rules ()
    ((catch tag . body)
     (let ((saved-catchers *active-catchers*))
       (unwind-protect
         (block label
           (set! *active-catchers*
                 (cons (cons tag (lambda (x) (return-from label x)))
                       *active-catchers*) )
           . body )
         (set! *active-catchers* saved-catchers) ) ) ) ) )
```

Что бы ни случилось, теперь `*active-catchers*` будет иметь корректное состояние при выходе из тела формы. Форму `block` можно использовать внутри `catch` не опасаясь, что `catch` не удалится из `*active-catchers*`, так как теперь за этим следит `unwind-protect`. Это гораздо лучше, хотя всё ещё не идеально: `*active-catchers*` доступна не только `catch` и `throw`, так что её состояние всё равно можно исказить (случайно или намеренно).

Форма `unwind-protect` обеспечивает защиту системы от противоречий, выполняя определённые действия после завершения вычислений. Следовательно, эта форма обязана знать, когда именно они завершаются. Но в присутствии продолжений с неограниченным временем жизни `unwind-protect` не может легко ответить на этот вопрос.[10]

Как мы уже не раз говорили, семантика управляющих структур далека от точного определения. Рассмотрим лишь несколько примеров, где нельзя однозначно сказать, какой мы получим результат:

```
(block foo
  (unwind-protect (return-from foo 1)
    (return-from foo 2) ) )                          → ?

(catch 'bar
  (block foo
    (unwind-protect (return-from foo (throw 'bar 1))
      (throw 'something (return-from foo 2)) ) ) )   → ?
```

Конечно, стремление к более точным определениям управляющих структур естественно, однако нельзя игнорировать очевидную неопределённость, привносимую продолжениями в понятие «после вычислений». Не всё можно выяснить только по исходному коду. Продолжения по определению *динамичны*, так как являются воплощением потока исполнения. Рассмотрим следующий пример:

[10] Правильно работающий аналог `unwind-protect` для Scheme — `dynamic-wind`, — был описан ещё в [FWH92]; см. также [Que93c].

```
(block bar
  (unwind-protect (return-from bar 1)
    (block foo π) ) )
```

`unwind-protect` вклинивается в поток исполнения и не даёт завершить переход, который выполняется в охраняемой ей форме. Вместо этого данный переход становится продолжением формы `(block foo ...)`. Если она просто вернёт результат, то это продолжение активируется и форма `(block bar ...)` передаст `1` своему продолжению. Если же внутри π будет выполнен переход, то данное продолжение должно быть отброшено и заменено продолжением перехода. В этом случае из-за `unwind-protect` «после вычислений `(return-from bar 1)`» не наступает вообще. (Мы обсудим этот феномен позже вместе деталями реализации данной формы.)

Конечно же, есть и другие управляющие формы. Особенно их жалует Common Lisp, в котором реализована даже старая `prog`, только под названием `tagbody`. Её можно легко проэмулировать с помощью `labels` и `block`. [**см. упр. 3.3**] Интересным фактом является то, что если продолжения имеют исключительно динамическое время жизни, то для реализации любого управления потоком исполнения достаточно форм `block`, `return-from` и `unwind-protect`. Аналогично, для продолжений с неограниченным временем жизни достаточно одной `call/cc`. Очевидно, что мы не сможем легко реализовать `call/cc`, имея лишь продолжения с динамическим временем жизни. Обратное вполне возможно, хотя это и стрельба из пушки по воробьям. Способ станет вполне очевидным после рассмотрения реализации интерпретатора с явными продолжениями.

Защита и динамические переменные

Некоторые реализации Scheme обеспечивают динамическое время жизни переменных не так, как мы показывали ранее. Они делают это с помощью `unwind-protect` или аналогичного механизма. Идея состоит в том, чтобы «одолжить» нужную лексическую переменную, восстановив впоследствии её значение обратно. Подобные динамические переменные реализуются с помощью формы `fluid-let`:

```
(fluid-let ((x α)) ≡ (let ((tmp x))
  β... )               (set! x α)
                       (unwind-protect
                         (begin β...)
                         (set! x tmp) ) )
```

В процессе вычисления β будет видна переменная `x` со значением α; предыдущее значение `x` сохраняется на время вычислений в локальной переменной *tmp* и восстанавливается после их завершения. Это подразумевает, что есть такая лексическая переменная `x`, которой можно воспользоваться. Обычно она глобальная, чтобы её было видно отовсюду. Если она будет локальной, то её

поведение будет (значительно) отличаться от должного поведения динамической переменной в Common Lisp: ведь тогда она будет правильно работать внутри `fluid-let`, но не в связывающих формах, вложенных во `fluid-let`. Далее, очевидно, что такие переменные тоже не дружат с `call/cc`. В итоге получается нечто ещё более хитрое, нежели обычные динамические переменные Common Lisp.

3.2. Участники вычислений

Сейчас мы считаем, что для проведения вычислений необходимы три вещи: выражение, окружение и продолжение. Тактическая цель вычислений: определить значение выражения в окружении. Стратегическая — передать это значение продолжению.

Мы определим новый интерпретатор, чтобы показать, какие продолжения нужны на каждом этапе вычислений. Так как обычно продолжения представляются снимками фреймов стека (или записей активаций), то мы будем использовать объекты для представления этих сущностей внутри разрабатываемого интерпретатора.

3.2.1. Краткий обзор объектов

В этом разделе мы не будем детально разбирать устройство объектной системы, отложив эту задачу до одиннадцатой главы. Здесь рассматриваются лишь три макроса и несколько правил именования. Такие макросы выражают суть объектов и в том или ином виде присутствуют в любой объектной системе любого языка. Объекты же используются для того, чтобы подсказать удобный вариант реализации продолжений. Уж очень хорошо понятие записи активации, инкапсулирующей различные данные, связанные с вызовами подпрограмм, укладывается в концепцию объектов с полями. Также у нас будет в распоряжении наследование, которое поможет вынести общие части реализации за скобки, уменьшая таким образом размер интерпретатора.

Я полагаю, что вы знакомы с философией, терминологией и подходами объектно-ориентированного программирования, так что будет достаточно показать, как здесь записываются известные вам идиомы, которые мы будем использовать.

Объекты группируются в *классы*; объекты одного класса имеют одинаковые *методы*; сообщения посылаются с помощью *обобщённых функций*, популяризованных Common Loops [BKK+86], CLOS [BDG+88] и ТЕЛОΣ [PNB93]. Для нас важнейшей возможностью объектно-ориентированного программирования является отделение обработки различных специальных форм и примитивных функций от ядра интерпретатора. Но всё имеет свою цену: в этом случае будет сложнее увидеть картину целиком, так как обработка будет размазана по нескольким местам.

Определение классов

Классы определяется с помощью `define-class` следующим образом:

```
(define-class класс суперкласс
  (поля...) )
```

Эта форма определяет класс с именем *класс*, который наследует поля и методы *суперкласса*, а также имеет свои собственные *поля*. Вместе с классом создаётся набор вспомогательных функций. Функция `make-`*класс* создаёт объекты этого класса; количество и порядок её аргументов соответствуют порядку указания полей при определении класса. Названия аксессоров чтения состоят из имени класса и имени поля, разделённых дефисом. Названия аксессоров записи аналогичны аксессорам чтения, только с `set-` в начале и восклицательным знаком в конце. Возвращаемое значение аксессоров записи не определено. Предикат *класс*`?` проверяет, является ли объект экземпляром данного класса.

Корнем иерархии наследования является класс `Object`, не имеющий полей.

Например, определение

```
(define-class continuation Object (k))
```

создаст следующие функции:

```
(make-continuation k)          ; конструктор
(continuation-k c)             ; аксессор чтения
(set-continuation-k! c k)      ; аксессор записи
(continuation? k)              ; предикат принадлежности
```

Определение обобщённых функций

Обобщённые функции определяются следующим образом:

```
(define-generic (функция аргументы)
  [трактовка-по-умолчанию...])
```

Эта форма определяет обобщённую *функцию*; формы *трактовки-по-умолчанию* станут её телом, если при вызове функции не найдётся подходящего специализированного варианта. Список аргументов указывается как обычно, за исключением того, что один из них является *дискриминантом*; дискриминант записывается в скобках:

```
(define-generic (invoke (f) v* r k)
  (wrong "Not a function" f r k) )
```

Таким образом определяется обобщённая функция `invoke`, для которой можно в последующем задать специализированные варианты. Данная функция имеет четыре аргумента, первый из них — `f` — это дискриминант. Если для класса `f` не найдётся специализированного варианта функции (метода класса `f`), то будет выбран вариант по умолчанию: вызов `wrong`.

Определение методов

Форма `define-method` используется для специализации обобщённых функций конкретными методами.

`(define-method (`*функция аргументы*`)`
  тело`... )`

Аргументы указываются аналогично `define-generic`. Класс дискриминанта, для которого создаётся метод, указывается после него. Например, мы можем создать метод `invoke` для класса `primitive` следующим образом:

```
(define-method (invoke (f primitive) v* r k)
  ((primitive-address f) v* r k) )
```

На этом мы заканчиваем обзор объектной системы и переходим к написанию интерпретатора. Детали реализации, а также другие возможности объектов будут рассмотрены в одиннадцатой главе. Здесь мы ограничимся наиболее простыми и известными из них, чтобы облегчить понимание и уменьшить количество возможных проблем.

3.2.2. Интерпретатор

Функция `evaluate` имеет три аргумента: выражение, окружение и продолжение. Начинает она свою работу с выяснения смысла выражения, чтобы выбрать правильный метод его вычисления, который хранится в специализированной функции. Перед тем, как продолжить, давайте договоримся о правилах именования переменных, которых теперь будет довольно много. Первое правило: сущность «список x» будем называть x`*`. Второе: сущности интерпретатора будем называть одной-двумя буквами для краткости:

`e`, `et`, `ec`, `ef`	выражения, формы
`r`	окружения
`k`, `kk`	продолжения
`v`	значения (числа, пары, замыкания и т. д.)
`f`	функции
`n`	идентификаторы

Всё, теперь принимаемся за интерпретатор. Для простоты он считает все атомы, кроме переменных, автоцитированными значениями.

```
(define (evaluate e r k)
  (if (atom? e)
      (cond ((symbol? e) (evaluate-variable e r k))
            (else        (evaluate-quote e r k)) )
      (case (car e)
        ((quote)  (evaluate-quote (cadr e) r k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r k))
```

```
((begin)  (evaluate-begin (cdr e) r k))
((set!)   (evaluate-set! (cadr e) (caddr e) r k))
((lambda) (evaluate-lambda (cadr e) (cddr e) r k))
(else     (evaluate-application (car e) (cdr e) r k)) ) ) )
```

Собственно интерпретатор состоит из трёх функций: `evaluate`, `invoke` и `resume`. Две последние являются обобщёнными и знают, как вызывать вызываемое и продолжать продолжаемое. Все вычисления в конечном счёте сводятся к обмену значениями между этими функциями. Вдобавок мы введём ещё две полезные обобщённые функции для работы с переменными: `lookup` и `update!`.

```
(define-generic (invoke (f) v* r k)
  (wrong "Not a function" f r k) )

(define-generic (resume (k continuation) v)
  (wrong "Unknown continuation" k) )

(define-generic (lookup (r environment) n k)
  (wrong "Not an environment" r n k) )

(define-generic (update! (r environment) n k v)
  (wrong "Not an environment" r n k) )
```

Все сущности, которыми мы будем оперировать, наследуются от трёх базовых классов:

```
(define-class value        Object ())
(define-class environment  Object ())
(define-class continuation Object (k))
```

Классы значений являются наследниками `value`, классы окружений — наследники `environment`, классы продолжений — `continuation`.

3.2.3. Цитирование

Специальная форма цитирования всё так же является наиболее простой, её задача сводится к передаче значения в неизменной форме текущему продолжению:

```
(define (evaluate-quotation v r k)
  (resume k v) )
```

3.2.4. Ветвление

Условный оператор использует два продолжения: текущее и продолжение вычисления условия, которое выберет и вычислит необходимую ветку. Для этого продолжения мы создадим отдельный класс. После вычисления условия

ещё остаётся вычисление той или иной ветки, а значит, в продолжении необходимо хранить сами ветки и окружение для их вычисления. Результат вычисления одной из веток надо будет передать продолжению условной формы, которое тоже надо где-то хранить. Таким образом, мы пишем:

```
(define-class if-cont continuation (et ef r))

(define (evaluate-if ec et ef r k)
  (evaluate ec r (make-if-cont k et ef r)) )

(define-method (resume (k if-cont) v)
  (evaluate (if v (if-cont-et k) (if-cont-ef k))
            (if-cont-r k)
            (if-cont-k k) ) )
```

Форма вначале вычисляет условие `ec` в своём окружении `r`, но с новым продолжением. Как только мы заканчиваем вычислять условие, результат передаётся `resume`, которая вызывает специализацию для нашего класса продолжений. В этом продолжении мы выполняем собственно выбор, вычисляем одну их сохранённых веток в сохранённом окружении и передаём результат сохранённому продолжению всей условной формы.[11]

3.2.5. Последовательность

Здесь нам тоже потребуются два продолжения: текущее и продолжение вычисления оставшихся форм.

```
(define-class begin-cont continuation (e* r))

(define (evaluate-begin e* r k)
  (if (pair? e*)
    (if (pair? (cdr e*))
        (evaluate (car e*) r (make-begin-cont k e* r))
        (evaluate (car e*) r k) )
    (resume k empty-begin-value) ) )

(define-method (resume (k begin-cont) v)
  (evaluate-begin (cdr (begin-cont-e* k))
                  (begin-cont-r k)
                  (begin-cont-k k) ) )
```

Случаи `(begin)` и `(begin` π`)` тривиальны. Если же `begin` передано больше выражений, то вычисление первого из них продолжается `(make-begin-cont`

[11] С точки зрения реализации можно считать, что `make-if-cont` кладёт в стек `et` и `ef`, а также `r`; под ними в стеке лежат аналогичные группы выражений и окружений, которые фактически и есть ничем иным, как продолжением `k`. А вызовы вроде `(if-cont-et k)` лишь снимают с верхушки стека нужные данные.

`k e* r)`. Это продолжение принимает значение `v` с помощью `resume`, игнорирует его и продолжает оставшиеся вычисления в том же окружении и с тем же продолжением.[12]

3.2.6. Окружения

Значения переменных хранятся в окружениях. Они тоже представляются объектами:

```
(define-class null-env environment ())
(define-class full-env environment (others name))
(define-class variable-env full-env (value))
```

Нам потребуются два типа окружений: пустое начальное окружение `null-env` и окружения с переменными `variable-env`. Последние хранят одну привязку имени `name` к значению `value`, а также ссылку на остальные привязки этого окружения в поле `others`. То есть это обычный A-список, разве что для хранения каждой привязки используется объект с тремя полями, а не две точечных пары.

Для нахождения значения переменной мы делаем следующее:

```
(define (evaluate-variable n r k)
  (lookup r n k) )

(define-method (lookup (r null-env) n k)
  (wrong "Unknown variable" n r k) )

(define-method (lookup (r full-env) n k)
  (lookup (full-env-others r) n k) )

(define-method (lookup (r variable-env) n k)
  (if (eqv? n (variable-env-name r))
      (resume k (variable-env-value r))
      (lookup (variable-env-others r) n k) ) )
```

Обобщённая функция `lookup` проходит по окружению, пока не найдёт подходящую привязку: с совпадающим именем и хранящую значение переменной. Найденное значение передаётся исходному продолжению с помощью `resume`.

Изменение значения происходит похожим образом:

```
(define-class set!-cont continuation (n r))

(define (evaluate-set! n e r k)
  (evaluate e r (make-set!-cont k n r)) )
```

[12] Внимательный читатель наверняка заметил странную форму `(cdr (begin-cont-e* k))` в методе `resume`. Конечно, мы могли бы отбросить уже вычисленное выражение ещё в `evaluate-begin`: `(make-begin-cont k (cdr e*) r)`, и получить тот же результат. Причина такого решения в том, что если случится ошибка, то у нас будет на руках её источник.

```
(define-method (resume (k set!-cont) v)
  (update! (set!-cont-r k) (set!-cont-n k) (set!-cont-k k) v) )

(define-method (update! (r null-env) n k v)
  (wrong "Unknown variable" n r k) )

(define-method (update! (r full-env) n k v)
  (update! (full-env-others r) n k v) )

(define-method (update! (r variable-env) n k v)
  (if (eqv? n (variable-env-name r))
      (begin (set-variable-env-value! r v)
             (resume k v) )
      (update! (variable-env-others r) n k v) ) )
```

Нам потребовалось вспомогательное продолжение, так как присваивание проходит в два этапа: сначала надо вычислить присваиваемое значение, потом присвоить его переменной. Класс `set!-cont` представляет необходимые продолжения, его метод `resume` лишь вызывает `update!` для установки значения, после чего продолжает дальнейшие вычисления.

3.2.7. Функции

Создать функцию легко, с этим справится `make-function`:

```
(define-class function value (variables body env))

(define (evaluate-lambda n* e* r k)
  (resume k (make-function n* e* r)) )
```

Чуть сложнее будет вызвать созданную функцию. Обратите внимание на неявное использование `progn`/`begin` для тела функций.

```
(define-method (invoke (f function) v* r k)
  (let ((env (extend-env (function-env f)
                         (function-variables f)
                         v* )))
    (evaluate-begin (function-body f) env k) ) )
```

Может показаться странным, что функция принимает текущее окружение `r`, но никак не использует его. Это сделано по нескольким причинам. Во-первых, обычно при компиляции текущие окружение и продолжение считаются чем-то вроде глобальных динамических переменных и передаются через жёстко заданные регистры, которые никак не выкинуть из реализации. Во-вторых, некоторые функции (о них поговорим позже, когда будем рассматривать рефлексию) могут изменять текущее окружение; например, отладочные функции по запросу пользователя могут изменять значения произвольных переменных.

Следующая функция расширяет окружение переменных. И выполняет проверку согласованности количества имён и связываемых с ними значений.

```
(define (extend-env env names values)
  (cond ((and (pair? names) (pair? values))
         (make-variable-env
          (extend-env env (cdr names) (cdr values))
          (car names)
          (car values) ) )
        ((and (null? names) (null? values)) env)
        ((symbol? names) (make-variable-env env names values))
        (else (wrong "Arity mismatch")) ) )
```

Осталось только определить собственно применение функций. Здесь надо помнить о том, что функция применяется к списку аргументов.

```
(define-class evfun-cont    continuation (e* r))
(define-class apply-cont    continuation (f  r))
(define-class argument-cont continuation (e* r))
(define-class gather-cont   continuation (v))

(define (evaluate-application e e* r k)
  (evaluate e r (make-evfun-cont k e* r)) )

(define-method (resume (k evfun-cont) f)
  (evaluate-arguments (evfun-cont-e* k)
                      (evfun-cont-r k)
                      (make-apply-cont (evfun-cont-k k) f
                                       (evfun-cont-r k) ) ) )
(define no-more-arguments '())

(define (evaluate-arguments e* r k)
  (if (pair? e*)
      (evaluate (car e*) r (make-argument-cont k e* r))
      (resume k no-more-arguments) ) )

(define-method (resume (k argument-cont) v)
  (evaluate-arguments (cdr (argument-cont-e* k))
                      (argument-cont-r k)
                      (make-gather-cont (argument-cont-k k) v) ) )

(define-method (resume (k gather-cont) v*)
  (resume (gather-cont-k k) (cons (gather-cont-v k) v*)) )

(define-method (resume (k apply-cont) v)
  (invoke (apply-cont-f k) v
          (apply-cont-r k)
          (apply-cont-k k) ) )
```

На первый взгляд, здесь всё слишком сложно, но только на первый. Вычисления проводятся слева направо, так что первой вычисляется сама функция с продолжением `evfun-cont`. Это продолжение должно вычислить аргументы функции и передать их продолжению, которое применит функцию к списку значений аргументов. В процессе вычисления аргументов мы обращаемся к продолжениям `gather-cont`, которые последовательно собирают вычисленные аргументы в список.

Давайте рассмотрим на примере, что происходит при вычислении `(cons foo bar)`. Пусть переменная `foo` имеет значение 33, а `bar` равна −77. Стек продолжений показан справа, а вычисляемое выражение слева. k — это текущее продолжение, r — текущее окружение. Функция-значение переменной `cons` записывается как *cons*.

evaluate `(cons foo bar)` r						k
evaluate `cons` r				evfun-cont `(foo bar)` r		k
resume *cons*				evfun-cont `(foo bar)` r		k
evaluate-arguments `(foo bar)` r				apply-cont *cons*		k
evaluate `foo` r		argument-cont `(foo bar)` r		apply-cont *cons*		k
resume 33		argument-cont `(foo bar)` r		apply-cont *cons*		k
evaluate-arguments `(bar)` r			gather-cont 33	apply-cont *cons*		k
evaluate `bar` r	argument-cont `()` r		gather-cont 33	apply-cont *cons*		k
resume −77	argument-cont `()` r		gather-cont 33	apply-cont *cons*		k
evaluate-arguments `()` r	gather-cont −77		gather-cont 33	apply-cont *cons*		k
resume ( )	gather-cont −77		gather-cont 33	apply-cont *cons*		k
resume (−77)			gather-cont 33	apply-cont *cons*		k
resume (33 −77)				apply-cont *cons*		k
invoke *cons* (33 −77)						k

3.3. Инициализация интерпретатора

Перед погружением в сокровенные тайны устройства управляющих форм, давайте сначала подготовим наш интерпретатор к запуску. Этот раздел похож на раздел 1.7. [**см. стр.** 45] Неплохо было бы вначале научить наш интерпретатор нескольким полезным вещам вроде `car`, поэтому объявим пару макросов, которые помогут нам наполнить его глобальное окружение.

```
(define-syntax definitial
  (syntax-rules ()
    ((definitial name)
     (definitial name 'void) )
    ((definitial name value)
     (begin (set! r.init (make-variable-env r.init 'name value))
            'name ) ) ) )

(define-class primitive value (name address))
```

```
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value arity)
     (definitial name
       (make-primitive
        'name (lambda (v* r k)
                (if (= arity (length v*))
                    (resume k (apply value v*))
                    (wrong "Incorrect arity" 'name v*) ) ) ) ) ) ) )

(define r.init (make-null-env))

(defprimitive cons cons 2)
(defprimitive car car 1)
```

Создаваемые примитивные функции должны вызываться той же функцией `invoke`, которая обрабатывает обычные функции. Каждый примитив имеет два поля. Первое из них служит для упрощения отладки: оно хранит имя примитива. Естественно, это лишь подсказка, так как ничто не мешает в дальнейшем связать один и тот же примитив с разными именами.[13] Второе поле хранит «адрес» примитива, ссылку на соответствующую функцию языка реализации интерпретатора. В итоге примитивы вызываются с помощью `invoke` следующим образом:

```
(define-method (invoke (f primitive) v* r k)
  ((primitive-address f) v* r k) )
```

Для запуска нашего прекрасного интерпретатора остаётся лишь определить начальное продолжение-заглушку. Это продолжение будет печатать на экран всё, что ему передают.

```
(define-class bottom-cont continuation (f))

(define-method (resume (k bottom-cont) v)
  ((bottom-cont-f k) v) )

(define (chapter3-interpreter)
  (define (toplevel)
    (evaluate (read)
              r.init
              (make-bottom-cont 'void display) )
    (toplevel) )
  (toplevel) )
```

[13] Эта подсказка позволяет также копировать примитивы по значению: выражение `(begin (set! foo car) (set! car 3) foo)` возвращает `#<car>` — собственное имя примитива, связанного с глобальной переменной.

Заметьте, что мы могли бы легко написать похожий интерпретатор на истинно объектно-ориентированном языке, например на Smalltalk [GR83], получив заодно доступ к его хвалёному отладчику и среде разработки. Для полного счастья останется только добавить те несколько строчек, что открывают гору маленьких окошек с контекстными подсказками.

3.4. Реализация управляющих форм

Начнём с самой мощной формы — `call/cc`. Парадоксально, но факт: это самая простая форма, если смотреть на количество кода. Благодаря используемому нами объектному подходу и явному присутствию продолжений в интерпретаторе, преобразование их в полноценные объекты языка становится тривиальным.

3.4.1. Реализация `call/cc`

Функция `call/cc` берёт текущее продолжение k, превращает его в объект, понятный `invoke`, и применяет к нему свой аргумент — унарную функцию. Следующий код чуть ли не буквально записывает это определение:

```
(definitial call/cc
  (make-primitive
   'call/cc
   (lambda (v* r k)
     (if (= 1 (length v*))
         (invoke (car v*) (list k) r k)
         (wrong "Incorrect arity" 'call/cc v*) ) ) ) )
```

Хоть тут и немного строчек, всё же стоит кое-что объяснить. `call/cc` это функция, но мы определяем её с помощью `defprimitive`, так как это единственный способ для функции добраться до `k`. Переменная `call/cc` (это всё же Lisp_1) связывается с объектом класса `primitive`. Для вызова объектов этого класса необходим «адрес» функции, которому у нас соответствуют функции языка определения вида `(lambda (v* r k) ...)`. После проверки на арность первый аргумент `call/cc` применяется к захваченному продолжению. Само продолжение мы никак не трогаем, оно остаётся объектом языка определения. Так как сохранённые «сырые» продолжения могут быть впоследствии переданы `invoke` напрямую, то её надо научить обращаться с ними:

```
(define-method (invoke (f continuation) v* r k)
  (if (= 1 (length v*))
      (resume f (car v*))
      (wrong "Continuations expect one argument" v* r k) ) )
```

3.4.2. Реализация `catch`

Форма `catch` по-своему интересна, так как требует разительно иного подхода, нежели форма `block`, которую мы рассмотрим чуть позже. Как обычно, начнём с добавления анализа `catch` и `throw` в `evaluate`:

```
...
((catch) (evaluate-catch (cadr e) (cddr e) r k))
((throw) (evaluate-throw (cadr e) (caddr e) r k))
...
```

Здесь решено сделать `throw` специальной формой, а не функцией. В первую очередь с целью походить на Common Lisp. Далее определим правила обработки формы `catch`:

```
(define-class catch-cont   continuation (body r))
(define-class labeled-cont continuation (tag))

(define (evaluate-catch tag body r k)
  (evaluate tag r (make-catch-cont k body r)) )

(define-method (resume (k catch-cont) v)
  (evaluate-begin (catch-cont-body k)
                  (catch-cont-r k)
                  (make-labeled-cont (catch-cont-k k) v) ) )
```

Как видите, `catch` вычисляет первый аргумент (метку), связывает с ней своё продолжение, создавая таким образом помеченный блок, и, наконец, последовательно вычисляет формы, составляющие её тело. Когда продолжение этого блока получает значение, оно просто перебрасывает его сохранённому продолжению самой формы `catch`. Форма `throw` чуть более сложная:

```
(define-class throw-cont    continuation (form r))
(define-class throwing-cont continuation (tag cont))

(define (evaluate-throw tag form r k)
  (evaluate tag r (make-throw-cont k form r)) )

(define-method (resume (k throw-cont) tag)
  (catch-lookup k tag k) )

(define-generic (catch-lookup (k) tag kk)
  (wrong "Not a continuation" k tag kk) )

(define-method (catch-lookup (k continuation) tag kk)
  (catch-lookup (continuation-k k) tag kk) )

(define-method (catch-lookup (k bottom-cont) tag kk)
  (wrong "No associated catch" k tag kk) )
```

```
(define-method (catch-lookup (k labeled-cont) tag kk)
  (if (eqv? tag (labeled-cont-tag k))  ; внимание на компаратор
      (evaluate (throw-cont-form kk)
                (throw-cont-r kk)
                (make-throwing-cont kk tag k) )
      (catch-lookup (labeled-cont-k k) tag kk) ) )

(define-method (resume (k throwing-cont) v)
  (resume (throwing-cont-cont k) v) )
```

Форма `throw` вычисляет первый аргумент и пытается найти продолжение с совпадающей меткой. Если в процессе поиска она добирается до начального продолжения, то сигнализирует об ошибке. Если же нет, то вычисляется второй аргумент `throw` и его значение передаётся найденному продолжению. Но передаётся оно по-хитрому: через `throwing-cont`. Дело в том, что в процессе вычисления этого значения тоже может возникнуть переход. Если бы продолжением данного вычисления было продолжение, сохранённое в метке внешней формы `throw`, то вложенная форма `throw` начинала бы поиски `catch` так, как будто бы переход на внешнюю метку уже произошёл. Но это не так, так что поиск следует вести от текущей формы `throw`, потому и создаётся специальное промежуточное продолжение. В итоге, когда мы пишем:

```
(catch 2
  (* 7 (catch 1
         (* 3 (catch 2
                (throw 1 (throw 2 5)) )) )) )
```

то получаем `(* 7 3 5)`, а не `5`.

Кроме того, реализация `throw` как специальной формы позволяет отлавливать больше ошибок.

```
(catch 2 (* 7 (throw 1 (throw 2 3))))
```

Эта форма гарантированно вернёт не `3`, а ошибку `"No associated catch"`, так как действительно нет `catch` с меткой `1` и не важно, что она вроде бы как не используется.

3.4.3. Реализация `block`

Для реализации лексических меток переходов необходимо решить две проблемы. Первая: гарантировать динамическое время жизни продолжений. Вторая: обеспечить лексическую видимость меток. Для решения второй задачи мы, естественно, воспользуемся лексическими окружениями, где лексическая область видимости есть «из коробки». Над первой же придётся немного поработать самостоятельно. Чтобы не было путаницы, у `block` будет личный класс окружений для хранения привязок меток к продолжениям.

Начинаем как обычно: добавляем распознавание формы `block` в `evaluate` и описываем необходимые функции-обработчики.

```
(define-class block-cont continuation (label))
(define-class block-env full-env (cont))

(define (evaluate-block label body r k)
  (let ((k (make-block-cont k label)))
    (evaluate-begin body
                    (make-block-env r label k)
                    k ) ) )

(define-method (resume (k block-cont) v)
  (resume (block-cont-k k) v) )
```

С нормальным поведением закончили, переходим к `return-from`. Сначала добавляем её в `evaluate`:

```
...
((block)       (evaluate-block (cadr e) (cddr e) r k))
((return-from) (evaluate-return-from (cadr e) (caddr e) r k))
...
```

Затем описываем обработку:

```
(define-class return-from-cont continuation (r label))

(define (evaluate-return-from label form r k)
  (evaluate form r (make-return-from-cont k r label)) )

(define-method (resume (k return-from-cont) v)
  (block-lookup (return-from-cont-r k)
                (return-from-cont-label k)
                (return-from-cont-k k)
                v ) )

(define-generic (block-lookup (r) n k v)
  (wrong "Not an environment" r n k v) )

(define-method (block-lookup (r block-env) n k v)
  (if (eq? n (block-env-name r))
      (unwind k v (block-env-cont r))
      (block-lookup (block-env-others r) n k v) ) )

(define-method (block-lookup (r full-env) n k v)
  (block-lookup (variable-env-others r) n k v) )

(define-method (block-lookup (r null-env) n k v)
  (wrong "Unknown block label" n r k v) )
```

```
(define-generic (unwind (k) v ktarget))

(define-method (unwind (k continuation) v ktarget)
  (if (eq? k ktarget)
      (resume k v)
      (unwind (continuation-k k) v ktarget) ) )

(define-method (unwind (k bottom-cont) v ktarget)
  (wrong "Obsolete continuation" v) )
```

После вычисления необходимого значения функция `block-lookup` отправляется на поиски продолжения, связанного с меткой `tag` в лексическом окружении формы `return-from`. Если такое продолжение существует, то дальше с помощью `unwind` она убеждается в том, что оно является частью текущего продолжения.

Поиск именованного блока, хранящего нужное продолжение, реализуется обобщённой функцией `block-lookup`. Она обучена пропускать ненужные окружения с обычными переменными, останавливаясь только на экземплярах `block-env`, хранящих нужные нам `block-cont`. Аналогично и `lookup` пропускает экземпляры `block-env`, останавливаясь лишь на `variable-env`. Именно с этой целью данные классы наследуются от общего предка: `full-env`. Это позволяет безболезненно добавлять новые классы окружений, которые не будут мешать уже существующим.

Наконец, обобщённая функция `unwind` передаёт вычисленное значение найденному продолжению, но только если оно ещё актуально — то есть доступно из текущего продолжения.

3.4.4. Реализация `unwind-protect`

Форма `unwind-protect` является самой сложной для реализации; нам понадобится изменить определения форм `catch` и `block`, чтобы они вели себя правильно, когда находятся внутри `unwind-protect`. Это хороший пример возможности, чьё введение требует переработки всего, что уже написано до этого. Но отсутствие `unwind-protect` приводит к другим сложностям в будущем, так что оно того стоит.

Начнём с определения поведения самой формы `unwind-protect` (которая, как мы уже говорили, мало чем отличается от `prog1`):

```
(define-class unwind-protect-cont continuation (cleanup r))
(define-class protect-return-cont continuation (value))

(define (evaluate-unwind-protect form cleanup r k)
  (evaluate form r
            (make-unwind-protect-cont k cleanup r) ) )
```

```
(define-method (resume (k unwind-protect-cont) v)
  (evaluate-begin (unwind-protect-cont-cleanup k)
                  (unwind-protect-cont-r k)
                  (make-protect-return-cont
                   (unwind-protect-cont-k k) v ) ) )

(define-method (resume (k protect-return-cont) v)
  (resume (protect-return-cont-k k) (protect-return-cont-value k)) )
```

Далее необходимо доработать `catch` и `block`, чтобы они выполняли действия, предписанные `unwind-protect`, даже в случае выхода из них с помощью `throw` или `return-from`. Для `catch` необходимо изменить обработку `throwing-cont`:

```
(define-method (resume (k throwing-cont) v)
  (unwind (throwing-cont-k k) v (throwing-cont-cont k)) )
```

И научить `unwind` выполнять сохранённые действия в процессе обхода стека:

```
(define-class unwind-cont continuation (value target))

(define-method (unwind (k unwind-protect-cont) v target)
  (evaluate-begin (unwind-protect-cont-cleanup k)
                  (unwind-protect-cont-r k)
                  (make-unwind-cont
                   (unwind-protect-cont-k k) v target ) ) )

(define-method (resume (k unwind-cont) v)
  (unwind (unwind-cont-k k)
          (unwind-cont-value k)
          (unwind-cont-target k) ) )
```

Теперь, чтобы передать значение при переходе, нам недостаточно просто его отдать нужному продолжению. Нам необходимо подняться по стеку продолжений с помощью `unwind` (*раскрутить* стек) от текущего до целевого продолжения, выполняя по пути соответствующую уборку. Продолжения форм-уборщиков имеют тип `unwind-cont`. Их обработка с помощью `resume` вызывает продолжение уборки до достижения цели на случай вложенных форм `unwind-protect`, а также устанавливает правильное продолжение на случай переходов внутри самих форм-уборщиков (тот самый процесс отбрасывания продолжений, который рассматривался на странице 113).

Что касается `block`, то тут даже делать ничего не надо. Как вы помните, `block-lookup` уже вызывает `unwind` для раскрутки стека с целью проверки актуальности перехода:

```
(define-method (block-lookup (r block-env) n k v)
  (if (eq? n (block-env-name r))
      (unwind k v (block-env-cont r))
      (block-lookup (block-env-others r) n k v) ) )
```

Так что остаётся только сказать спасибо обобщённым функциям.

Может показаться, что с появлением `unwind-protect` форма `block` перестала быть быстрее `catch`, ведь они обе вынуждены пользоваться медленной `unwind`. В общем случае, конечно, да, но в частностях, коих большинство, это не так: `unwind-protect` является специальной формой, так что она не может быть спутана с обычной функцией, её всегда надо использовать явно. А если `return-from` прямо видит метку соответствующего `block` (то есть когда между ними нет `lambda`- или `unwind-protect`-форм), то `unwind` будет работать так же быстро, как и раньше.

В Common Lisp (CLtL2 [Ste90]) присутствует ещё одно интересное ограничение, касающееся переходов из форм-уборщиков. Эти переходы не могут вести внутрь той формы, из которой в теле `unwind-protect` был вызван выход. Введено такое ограничение с целью недопущения бесконечных циклов из переходов, любые попытки выбраться из которых пресекаются `unwind-protect`. [**см. упр. 3.9**] Следовательно, следующая программа выдаст ошибку, так как форма-уборщик хочет прыгнуть ближе, чем прыжок на `1`, который уже в процессе.

Common Lisp
```
(catch 1
  (catch 2
    (unwind-protect (throw 1 'foo)
      (throw 2 'bar) ) ) )          → ошибка!
```

3.5. Сравнение `call/cc` и `catch`

Благодаря объектам, продолжения можно представлять связным списком блоков. Некоторые из этих блоков доступны прямо в лексическом окружении; до других необходимо пробираться, проходя через несколько промежуточных продолжений; третьи вызывают выполнение определённых действий, когда через них проходят.

В языках вроде Лиспа, где есть продолжения с динамическим временем жизни, стек вызовов и продолжения являются синонимами. Когда мы пишем `(evaluate ec r (make-if-cont k et ef r))`, мы явно кладём в стек блок кода, который будет обрабатывать значение, которое вернёт условие `if`-формы. И наоборот, когда мы пишем `(evaluate-begin (cdr (begin-cont-e* k)) (begin-cont-r k) (begin-cont-k k))`, то это значит, что текущий блок `k` надо выбросить и поставить на его место `(begin-cont-k k)`. Можно легко убедиться в том, что такие блоки действительно выбрасываются, в стеке не остаются недовыполненные куски продолжений. Таким образом, когда мы выходим из блока, все продолжения, указывающие на него и, возможно, сохранённые в других блоках, становятся недействительными. Обычно продолжения неявно хранятся в стеке или даже в нескольких стеках, согласованных

между собой, а переходы между ними компилируются в примитивы языка Си: `setjmp`/`longjmp`. [**см. стр. 478**]

В диалекте EuLisp [PE92] есть специальная форма `let/cc` со следующим синтаксисом:

`(let/cc` *переменная формы*`...)` EuLisp

В диалекте Dylan [App92b] тоже есть подобная форма:

`(bind-exit (`*переменная*`)` *формы*`...)` Dylan

Эта форма связывает текущее продолжение с *переменной*, имеющей область видимости, ограниченную телом `let/cc` или `bind-exit`. В этом случае продолжение несомненно является полноценным объектом, имеющим интерфейс унарной функции. Но его *полезное* время жизни динамическое, его можно использовать лишь во время вычисления тела формы `let/cc` или `bind-exit`. Точнее, само продолжение, хранящееся в *переменной*, имеет неограниченное время жизни, но становится бесполезным при выходе из связывающей формы. Это характерная для EuLisp и Dylan черта, но её нет как в Scheme (где продолжения истинно неограниченны), так и в Common Lisp (где они вообще объекты второго класса). Тем не менее, такое поведение можно проэмулировать в Scheme:

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc variable . body)
     (block variable
       (let ((variable (lambda (x) (return-from variable x))))
         . body ) ) ) ) )
```

В мире Scheme продолжения больше нельзя считать неявной частью стека, так как они могут храниться во внешних структурах данных. Поэтому приходится применять другую модель: древовидную, которую иногда называют *стек-кактус* или *спагетти-стек*. Наиболее простой способ её реализовать: вообще не пользоваться аппаратным стеком, размещая все фреймы в куче.

Такой подход унифицирует выделение памяти под структуры данных и, по мнению [AS94], облегчает портирование. Тем не менее, он приводит к фрагментации, что вынуждает явно хранить ссылки между продолжениями. (Хотя в [MB93] приведено несколько вариантов решения этих проблем.) Как правило, ради эффективности в аппаратный стек стараются поместить максимум данных о ходе исполнения программы, так что каноническая реализация `call/cc` делает снимки стека и сохраняет в куче именно их; таким образом, продолжения — это как раз такие снимки стека. Конечно, существуют и другие варианты реализации, рассмотренные, например, в [CHO88, HDB90], где используются разделяемые копии, отложенное копирование, частичное копирование и т. д. Естественно, каждый из этих вариантов даёт свои преимущества, но за определённую плату.

Форма `call/cc` больше похожа на `block`, нежели на `catch`. Оба типа продолжений имеют лексическую область видимости, они различаются только временем жизни. В некоторых диалектах, вроде [IM89], есть урезанный вариант `call/cc`. Называется он `call/ep` (от *call with exit procedure*); эта *процедура выхода* хорошо видна в `block`/`return-from`, равно как и в `let/cc`. Интерфейс у `call/ep` такой же, как и у `call/cc`:

```
(call/ep (lambda (exit) ...))
```

Переменная `exit` унарной функции-аргумента связывается с продолжением формы `call/ep` на время вычисления тела этой функции. Схожесть с `block` налицо, разве что мы используем обычное окружение переменных, а не отдельное окружение лексических меток. Основное их отличие в том, что `call/ep` делает продолжение полноценным объектом, который можно использовать так же, как любой другой объект вроде чисел, замыканий или списков. Имея `block`, мы тоже можем создать функционально аналогичный объект, написав `(lambda (x) (return-from` *метка* `x))`. Но все возможные места выхода из `block` известны статически (это соответствующие формы `return-from`), тогда как в `call/ep` совсем по-другому: например, по выражению `(call/ep foo)` нельзя понять, может ли произойти переход или нет. Единственный способ это узнать — проанализировать `foo`, но эта функция может быть определена в совершенно другом месте, а то и вовсе генерироваться динамически. Следовательно, функция `call/ep` более сложна для компилятора, чем специальная форма `block`, но вместе с тем имеет и больше возможностей.

Продолжая сравнивать `call/ep` и `block`, мы замечаем больше отличий. Например, для формы `call/ep`, в которой аргумент записан в виде явной `lambda`-формы, можно не создавать замыкание. Следовательно, эффективный компилятор должен отделять случай `(call/ep (lambda ...))` от остальных. Это похоже на специальные формы, так как они тоже трактуются по-особенному. В Scheme принято использовать функции как основной инструмент построения абстракций, тогда как специальные формы являются чем-то вроде подсказок компилятору. Они часто одинаково мощны, вопрос лишь в балансе сложности — кому важнее облегчить жизнь: пользователю или, наоборот, разработчику языка.

Подводя итог, если вам нужна мощь за адекватную цену, то `call/cc` к вашим услугам, так как она позволяет реализовать все мыслимые управляющие конструкции: переходы, сопрограммы, частичные продолжения и так далее. Если же вам нужны только «нормальные» вещи (а Лисп уже не раз показывал, что можно писать удивительные программы и без `call/cc`), то используйте управляющие формы Common Lisp, простые и компилирующиеся в эффективный машинный код.

3.6. Продолжения в программировании

Существует стиль программирования, называемый «*стилем передачи продолжений*» (continuation passing style, CPS). В нём во главу угла ставится явное указание не только того, что возвращать в качестве результата функции, но и кому. После завершения вычислений функция не возвращает результат абстрактному получателю куда-то «наверх», а применяет конкретного получателя, представленного продолжением, к результату. В общем, если у нас есть вычисление `(foo (bar))`, то оно выворачивается наизнанку, преобразуясь в следующий вид: `(new-bar foo)`, где `foo` и является продолжением, которому `new-bar` передаст результат вычислений. Давайте рассмотрим данное преобразование на примере многострадального факториала. Пусть мы хотим вычислить $n(n!)$:

```
(define (fact n k)
  (if (= n 0) (k 1)
      (fact (- n 1) (lambda (r) (k (* n r)))) ) )

(fact n (lambda (r) (* n r))) → n(n!)
```

Факториал теперь принимает дополнительный аргумент `k`: получателя вычисленного факториала. Если результат равен единице, то к ней просто применяется `k`. Если же результат сразу сказать нельзя, то следует ожидаемый рекурсивный вызов. Проблема состоит в том, что хорошо было бы сначала умножить факториал $(\mathtt{n}-1)$ на `n` и только потом уже передавать произведение получателю, а форма `(k (* n (fact (- n 1) k)))` делает всё наоборот! Поэтому и мы всё сделаем шиворот-навыворот: пусть получатель сам умножает результат на `n`. Настоящий получатель оборачивается в функцию: `(lambda (r) (k (* n r)))`, и передаётся следующему рекурсивному вызову.

Такое определение факториала даёт возможность вычислять различные величины с помощью одного и того же определения. Например, обычный факториал: `(fact` n `(lambda (x) x))`, или удвоенный: `(fact` n `(lambda (x) (* 2 x)))`, или что-то более сложное.

3.6.1. Составные значения

Продолжения очень удобно использовать для обработки составных величин. Существуют вычисления, результатом которых является не одна величина, а несколько. Например, в Common Lisp целочисленное деление (`truncate`) одновременно возвращает частное и остаток. Пусть у нас тоже есть подобная функция — назовём её `divide`, — которая принимает два числа и продолжение, вычисляет частное и остаток от деления, а затем применяет переданное продолжение к этим величинам. Например, вот так можно проверить правильность выполнения деления этой функцией:

```
(let* ((p (read)) (q (read)))
```

```
(divide p q (lambda (quotient remainder)
              (= p (+ (* quotient q) remainder)) )) )
```

Менее тривиальный пример — вычисление коэффициентов Безу.[14] Соотношение Безу утверждает, что для любых целых чисел n и p можно найти такую пару целых u и v, что $un + vp = \text{НОД}(n, p)$. Для вычисления коэффициентов u и v можно использовать расширенный алгоритм Евклида.

```
(define (bezout n p k)   ; пусть n > p
  (divide
   n p (lambda (q r)
         (if (= r 0)
             (k 0 1)     ; т. к. 0 · qp + 1 · p = p
             (bezout
              p r (lambda (u v)
                    (k v (- u (* v q))) ) ) ) ) ) )
```

Функция `bezout` использует `divide`, чтобы сохранить в `q` и `r` частное и остаток от деления `n` на `p`. Если n делится нацело на p, то очевидно, что их наибольший общий делитель равен p и есть тривиальное решение: 0 и 1. Если остаток не равен нулю, то… попробуйте доказать правильность этого алгоритма самостоятельно; для этого не надо быть экспертом в теории чисел, достаточно знать свойства НОД. А здесь мы ограничимся простой проверкой:

```
(bezout 1991 1960 list) → (-569 578)
```

3.6.2. Хвостовая рекурсия

В примере с вычислением факториала с помощью продолжений вызов `fact` в конце концов приводил к ещё одному вызову `fact`. Если мы проследим за вычислением `(fact 3 list)`, то, отбрасывая очевидные шаги, получим следующую картину:

```
(fact 3 list)
≡ (fact 2 (lambda (r) (k (* n r))))
                                    | n → 3
                                    | k ≡ list
≡ (fact 1 (lambda (r) (k (* n r))))
                                    | n → 2
                                    | k → (lambda (r) (k (* n r)))
                                                          | n → 3
                                                          | k ≡ list
≡ (k (* n 1))
             | n → 2
             | k → (lambda (r) (k (* n r)))
                                           | n → 3
                                           | k ≡ list
```

[14] Фух! Наконец-то мне удалось опубликовать эту функцию! Она с 1981 года валяется у меня без дела.

```
≡ (k (* n 2))|
             |n → 3
             |k ≡ list
→ (6)
```

Когда `fact` вызывает `fact`, вторая функция вычисляется с тем же продолжением, что и первая. Такое явление называется *хвостовой рекурсией* — почему рекурсия, понятно, а хвостовая, потому что этот вызов выполняется в «хвосте» вычислений: сразу же после него следует выход из функции. Хвостовая рекурсия — это частный случай хвостового вызова. Хвостовой вызов происходит тогда, когда текущее вычисление может быть полностью заменено вызываемым. То есть вызов происходит из *хвостовой позиции*, если он выполняется с *неизменным продолжением*.

В примере с вычислением коэффициентов Безу функция `bezout` вызывает `divide` из хвостовой позиции. Функция `divide` вызывает своё продолжение из хвостовой позиции. Это продолжение рекурсивно вызывает `bezout` опять-таки из хвостовой позиции.

Но в классическом факториале `(* n (fact (- n 1)))` рекурсивный вызов `fact` происходит не из хвостовой позиции. Он *завёрнут* в продолжение, так как значение `(fact (- n 1))` ещё ожидается для умножения на `n`; вызов тут не является последней необходимой операцией, всё вычисление нельзя свести к нему.

Хвостовые вызовы позволяют отбрасывать ненужные окружения и фреймы стека, так как при таких вызовах они больше никогда не будут использоваться. Следовательно, их можно не сохранять, экономя таким образом драгоценную стековую память. Подобные оптимизации были детально изучены французским лисп-сообществом, что позволило существенно ускорить интерпретацию [Gre77, Cha80, SJ87]; см. также [Han90].

Оптимизация хвостовой рекурсии — это очень желанное свойство интерпретатора; не только для пользователя, но и для самого интерпретатора. Самое очевидное место, где она была бы полезной, — это форма `begin`. До сих пор она определялась следующим образом:

```
(define (evaluate-begin e* r k)
  (if (pair? e)
      (if (pair? (cdr e*))
          (evaluate (car e*) r (make-begin-cont k e* r))
          (evaluate (car e*) r k) )
      (resume k empty-begin-value) ) )

(define-method (resume (k begin-cont) v)
  (evaluate-begin (cdr (begin-cont-e* k))
                  (begin-cont-r k)
                  (begin-cont-k k) ) )
```

Заметьте, здесь каждый вызов является хвостовым. Также здесь используется одна небольшая оптимизация. Можно определить эту форму проще:

```
(define (evaluate-begin e* r k)
  (if (pair? e*)
      (evaluate (car e*) r (make-begin-cont k e* r))
      (resume k empty-begin-value) ) )

(define-method (resume (k begin-cont) v)
  (let ((e* (cdr (begin-cond-e* k))))
    (if (pair? e*)
        (evaluate-begin e* (begin-cont-r k) (begin-cont-k k))
        (resume (begin-cont-k k) v) ) ) )
```

Но первый вариант предпочтительнее, так как в этом случае при вычислении последнего оставшегося выражения мы не тратим время на создание лишнего продолжения `(make-begin-cont k e* r)`, которое фактически равно `k`, а сразу же переходим в нужное продолжение. Конечно, в Лиспе есть сборка мусора, но это не означает, что можно мусорить ненужными объектами на каждом шагу. Это небольшая, но важная оптимизация, ведь каждый `begin` когда-нибудь заканчивается!

Аналогично можно оптимизировать и вычисление аргументов функции, переписав его следующим образом:

```
(define-class no-more-argument-cont continuation ())

(define (evaluate-arguments e* r k)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (evaluate (car e*) r (make-argument-cont k e* r))
          (evaluate (car e*) r (make-no-more-argument-cont k)) )
      (resume k no-more-arguments) ) )

(define-method (resume (k make-no-more-argument-cont) v)
  (resume (no-more-argument-cont-k k) (list v)) )
```

Это новое продолжение, хранящее список из последнего вычисленного значения, избавляет нас от необходимости передавать окружение `r` целиком. Данный приём впервые использован Митчеллом Уондом и Дэниелом Фридманом в [Wan80b].

3.7. Частичные продолжения

Среди прочих вопросов, поднимаемых продолжениями, есть ещё один довольно интересный: что именно случается с отбрасываемым при переходе кодом? Другими словами, с тем куском продолжения (или стека), который находится между положениями до прыжка и после. Мы говорили, что такой

срез стека не сохраняется при переходе. Но он вовсе не является бесполезным: ведь если бы через него не перешагнули, то он бы принял какое-то значение, выполнил определённые действия и передал бы полученное значение своему продолжению. То есть вёл бы себя как обычная функция. Во многих работах, вроде [FFDM87, FF87, Fel88, DF90, HD90, QS91, MQ94], приводятся способы сохранения и приёмы использования этих срезов — *частичных продолжений* (partial/delimited continuations).

Рассмотрим следующий простой пример:

```
(+ 1 (call/cc (lambda (k) (set! foo k) 2))) → 3
(foo 3)                                     → 4
```

Какое именно продолжение хранится в `foo`? Казалось бы $\lambda u.1 + u$, но чему тогда равно `(foo (foo 4))`?

```
(foo (foo 4))                               → 5
```

Получается `5`, а не ожидаемое значение `6`, которое бы получилось при правильной композиции функций. Дело в том, что вызов продолжения означает отбрасывание всех последующих вычислений ради продолжения других вычислений. Таким образом, вызов продолжения внутри `foo` приводит к вычислению значения $\lambda u.1 + u$ при $u = 4$, которое становится значением всего выражения, и второй вызов `foo` вообще не происходит — он не нужен, ведь значение выражения уже вычислено и передано продолжению! Именно в этом проблема: мы захватили обычное продолжение, а не частичное. Обычные продолжения *активируются* и полностью заменяют стек собой, а *не вызываются* как функции.

Возможно, так будет понятнее. В `foo` мы сохранили `(+ 1 [])`. Это всё, что ещё осталось вычислить. Так как аргументы передаются по значению, то вычисление аргумента-продолжения в `(foo (foo 4))` фактически завершает вычисления, отбрасывает `(foo [])` и возвращает значение формы `(+ 1 4)`, которое, очевидно, равно `5`.

Частичные продолжения представляют собой лишь часть оставшихся вычислений, тогда как обычные продолжения — это *все* оставшиеся вычисления. В статьях [FWFD88, DF90, HD90, QS91] приводятся способы захвата частичных и, следовательно, поддающихся композиции продолжений. Предположим, теперь с `foo` связано продолжение `[(+ 1 [])]`, где внешние квадратные скобки означают, что оно ведёт себя как функция. Тогда `(foo (foo 4))` будет эквивалентно уже `(foo [(+ 1 [4])])`, что превращается в `(+ 1 5)`, которое в итоге даёт `6`. Захваченное продолжение `[(+ 1 [])]` определяет не все последующие вычисления, которые когда-либо произойдут, а только их часть вплоть до момента возврата значения. Для интерактивной сессии продолжением обычных продолжений является *главный цикл* (он же `toplevel`), именно ему продолжения передают своё значение, а он выводит его на экран, читает следующее выражение из входного потока, вычисляет его и так далее. Продолжение частичных продолжений неизвестно, именно поэтому они конечны

и ведут себя как обычные функции — ведь функции тоже не знают, кому они вернут значение.

Давайте взглянем на наш пример с `(set! foo k)` с другой стороны. Оставим всё по-прежнему, но объединим эти два выражения в явную последовательность:

```
(begin (+ 1 (call/cc (lambda (k) (set! foo k) 2)))
       (foo 3) )
```

Бабах! Мы получили бесконечный цикл, так как `foo` оказывается теперь связанной с `(begin (+ 1 []) (foo 3))`, что приводит к рекурсии. Как видим, главный цикл — это не только последовательное вычисление выражений. Если мы хотим правильно его проэмулировать, то вдобавок необходимо изменять продолжение каждого вычисляемого в главном цикле выражения:

```
(let (foo sequel print?)
  (define-syntax toplevel
    (syntax-rules ()
      ((toplevel e) (toplevel-eval (lambda () e))) ) )
  (define (toplevel-eval thunk)
    (call/cc (lambda (k)
               (set! print? #t)
               (set! sequel k)
               (let ((v (thunk)))
                 (when print? (display v) (set! print? #f))
                 (sequel v) ) )) )
  (toplevel (+ 1 (call/cc (lambda (k) (set! foo k) 2))))
  (toplevel (foo 3))
  (toplevel (foo (foo 4))) )
```

Каждый раз, когда мы хотим вычислить выражение с помощью `toplevel`, его продолжение — *продолжение* работы `toplevel` — сохраняется в переменной `sequel`. Любое продолжение, захватываемое внутри `thunk`, теперь будет ограничено текущей вычисляемой формой. Аналогичным образом применяя присваивание, можно сохранить любой срез стека в виде частичного продолжения. Как видим, все продолжения с неограниченным временем жизни для своего создания требуют побочных эффектов.

Частичные продолжения явно указывают, когда необходимо остановить вычисления. Этот эффект может быть полезен в некоторых случаях, а также интересен сам по себе. Мы вполне можем даже переписать нашу `call/cc` так, чтобы она захватывала именно частичные продолжения вплоть до `toplevel`. Естественно, кроме них потребуются также и переходы на тот случай, когда мы действительно не заинтересованы в сохранении срезов стека. Но, с другой стороны, частичные продолжения в реальности используются довольно редко; сложно привести пример программы, где частичные продолжения были бы действительно полезны, но при этом не усложняли бы её сильнее обычных.

Тем не менее, они важны как ещё один пример управляющей формы, которую можно реализовать на Scheme с помощью `call/cc` и присваивания.

3.8. Заключение

Продолжения вездесущи. Если вы понимаете продолжения, вы одновременно овладели ещё одним стилем программирования, получили широчайшие возможности управления ходом вычислений и знаете, во что вам обойдётся это управление. Продолжения тесно связаны с потоком исполнения, так как они динамически определяют всё, что ещё осталось сделать. Поэтому они так важны и полезны для обработки исключений.

Интерпретатор, определённый в этой главе, довольно мощный, но легко понятный только по частям. Это обычное дело для объектно-ориентированного стиля: есть много маленьких и простых кусочков, но не так просто составить понимание цельной картины того, как они работают вместе. Интерпретатор модульный и легко расширяется новыми возможностями. Он не особо быстрый, так как в процессе работы создаёт целую гору объектов, которые удаляются тут же после использования. Конечно, это является одной из задач компилятора: выяснить, какие из объектов действительно стоит создавать и сохранять.

3.9. Упражнения

Упражнение 3.1 Что вернёт `(call/cc call/cc)`? Зависит ли ответ от порядка вычислений?

Упражнение 3.2 А что вернёт `((call/cc call/cc) (call/cc call/cc))`?

Упражнение 3.3 Реализуйте пару `tagbody`/`go` с помощью `block`, `catch` и `labels`. Напомним синтаксис этой формы из Common Lisp:

```
(tagbody
          выражения0...
  метка1  выражения1...
          ...
  меткаi  выраженияi...
          ... )
```

Все *выраженияi* (и только они) могут содержать безусловные переходы `(go` *метка*`)` и возвраты `(return` *значение*`)`. Если `return` не будет, то форма `tagbody` возвращает `nil`.

Упражнение 3.4 Вы скорее всего заметили, что функции при вызове проверяют фактическую арность: количество переданных им аргументов. Измените механизм создания функций так, чтобы правильная арность рассчиты-

валась только один раз. Можете считать, что функции бывают только фиксированной арности.

Упражнение 3.5 Определите функцию `apply` для интерпретатора из этой главы.

Упражнение 3.6 Реализуйте поддержку функций переменной арности для интерпретатора из этой главы.

Упражнение 3.7 Измените функцию запуска интерпретатора так, чтобы она вызывала `evaluate` только единожды.

Упражнение 3.8 Способ реализации продолжений из раздела 3.4.1 отделяет продолжения от других значений. Поэтому мы вынуждены реализовывать метод `invoke` лично для класса продолжений, представляемых функциями языка определения. Переопределите `call/cc` так, чтобы она возвращала объекты определяемого языка, являющиеся экземплярами класса-наследника `value`, соответствующего продолжениям.

Упражнение 3.9 Напишите на Common Lisp функцию `eternal-return`, принимающую замыкание и вызывающую его в бесконечном цикле. Этот цикл должен быть истинно бесконечным: перекройте абсолютно все выходы из него.

Упражнение 3.10 Рассмотрим следующую хитроумную функцию (спасибо за неё Алану Бодену):

```
(define (make-box value)
  (let ((box
         (call/cc
          (lambda (exit)
            (letrec
             ((behavior
               (call/cc
                (lambda (store)
                  (exit (lambda (msg . new)
                          (call/cc
                           (lambda (caller)
                             (case msg
                               ((get) (store (cons (car behavior)
                                                   caller )))
                               ((set)
                                (store
                                 (cons (car new)
                                       caller ) ) ) ) ) )) ) ) ))
              ((cdr behavior) (car behavior)) ) ) ) ))
    (box 'set value)
    box ) )
```

Предположим, в `box1` лежит значение `(make-box 33)`, тогда что получится в результате следующих вычислений?

```
(box1 'get)
(begin (box1 'set 44) (box1 'get))
```

Упражнение 3.11 Среди всех наших функций только `evaluate` не является обобщённой. Можно создать класс программ, от которого будут наследоваться подклассы программ с различным синтаксисом. Правда, в этом случае мы не сможем хранить программы как S-выражения, они должны быть объектами. Соответственно, функция `evaluate` уже должна быть обобщённой. Это позволит легко вводить новые специальные формы (возможно, даже прямо из определяемого языка). Воплотите эту идею в жизнь.

Упражнение 3.12 Реализуйте оператор `throw` как функцию, а не специальную форму.

Упражнение 3.13 Сравните скорость выполнения обычного кода и переписанного в стиле передачи продолжений.

Упражнение 3.14 Реализуйте `call/cc` с помощью функции `the-current-continuation`, которая определяется следующим образом:

```
(define (the-current-continuation)
  (call/cc (lambda (k) k)) )
```

Рекомендуемая литература

Годный, нетривиальный пример использования продолжений приведён в [Wan80a]. Также стоит почитать [HFW84] об эмуляции сопрограмм. В [dR87] прекрасно рассказано о развитии понимания важности рефлексии для управляющих форм.

Глава 4

Присваивание и побочные эффекты

Предыдущие главы с их нескончаемыми повторениями и вариациями чем-то напоминают «Болеро» Мориса Равеля. Но один мотив, как вы могли заметить, в них отсутствовал: присваивание и побочные эффекты. Чистые языки их презирают из-за некоторых скверных особенностей, но поскольку в диалектах Лиспа их существование допускается, нам следует их изучить. В этой главе подробно рассматривается присваивание, а также прочие побочные эффекты. По мере нашего продвижения мы также будем вынуждены отвлекаться на смежные темы вроде равенства и семантики цитирования.

Присваивание, пришедшее из традиционных алгоритмических языков, позволяет в той или иной мере изменить значение, связанное с переменной. Это приводит к необходимости сохранять состояние программы (в том или ином виде), где будет записано, что такая-то переменная отныне имеет следующее значение. Для свободно владеющих императивными языками смысл присваивания кажется простым и очевидным. В этой главе мы покажем, как замыкания, а также наследие λ-исчисления, усложняют понятия связывания и переменных.

Главная проблема с присваиванием (и побочными эффектами) состоит в том, чтобы выбрать для их описания формальную теорию, которая не зависит от описываемых свойств. То есть определить язык с присваиванием и побочными эффектами, не используя для этого присваивание и побочные эффекты. Не то, чтобы мы их до этого использовали повсюду; единственные побочные эффекты в наших предыдущих интерпретаторах были сосредоточены в функции `update!` (естественно, для определения присваивания) и в определениях «хирургических инструментов» вроде `set-car!`, которые являлись лишь обёртками для родных функций языка определения.

4.1. Присваивание

Присваивание, как мы сказали, позволяет изменить значение переменной. Например, давайте напишем программу для определения минимального и

максимального элемента в двоичном дереве, используя две переменные для хранения наименьшего и наибольшего элемента из уже просмотренных. Мы получим что-то вроде следующего:

```
(define (min-max tree)
  (define (first-number tree)
    (if (pair? tree)
        (first-number (car tree))
        tree ) )
  (let* ((min (first-number tree))
         (max min) )
    (define (scan! tree)
      (cond ((pair? tree)
             (scan! (car tree))
             (scan! (cdr tree)) )
            (else (if (> tree max) (set! max tree)
                      (if (< tree min) (set! min tree)) )) ) )
    (scan! tree)
    (list min max) ) )
```

Функция `min-max` проста для понимания и требует лишь два числа и две точечные пары для работы. Алгоритм не особо отличается от того, который бы получился на Паскале, и использует переменные точно так же, как они используются в императивных языках. Но, что важно, побочные эффекты изменения значений этих локальных переменных не видны снаружи функции `min-max`, то есть не влияют на остальные части программы. Это пример доброкачественных побочных эффектов: программа становится более понятной и эффективной, чем написанная в чисто функциональном стиле. [**см. упр. 4.1**]

Присваивание локальным переменным, которые не используются совместно несколькими функциями, не вызывает существенных проблем. Например, следующая функция последовательно возвращает натуральные числа, начиная с нуля. Очевидно, что значением переменной `n` сразу после присваивания является только что присвоенное значение.

```
(define enumerate
  (let ((n -1))
    (lambda () (set! n (+ n 1))
               n ) ) )
```

Каждый вызов `enumerate` возвращает новое натуральное число. Функция `enumerate` имеет внутреннее состояние, представляемое переменной `n`, которая изменяется с каждым следующим вызовом функции. Но то, что `n` принадлежит замыканию, является проблемой. В λ-исчислении нет ни слова о присваивании. Ведь когда мы применяем функцию в математике (и, соответственно, в λ-исчислении), то просто заменяем в её теле все переменные соответствующими значениями аргументов. Из-за присваивания данная семантика подстановки становится неверной, а программы теряют ссылочную прозрачность.

Если подставить вместе переменной `n` её начальное значение в `enumerate`, то эта функция возвращала бы не натуральные числа, а исключительно `-1`. Таким образом, присваивание вынуждает нас отказаться от модели вычислений с мгновенной подстановкой значений. Теперь подстановка является отложенной и выполняется только тогда, когда мы явно потребуем значение. Соответственно, определённым образом расставленные присваивания могут изменить это значение до подстановки.

Но с расположением присваиваний есть свои проблемы. Рассмотрим следующую программу:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (set! set-winner! (lambda (new-name) (set! name new-name)
                                       name ))
  (set!-winner "Me")
  (winner) )
```

Что же вернёт `(winner)`: `"Nemo"` или `"Me"`? Другими словами, влияет ли присваивание внутри `set-winner!` на значение, возвращаемое `winner`?

И снова λ-исчисление хранит молчание. У него есть причины быть немногословным: ведь присваивание чуждо ему. Ладно, мы вроде бы говорили, что создание функции захватывает окружение определения; значит, функции `winner` и `set!-winner` по крайней мере знают, что в момент их создания переменная `name` имела значение `"Nemo"`. Также кажется очевидным, что `(set-winner! "Me")` вернёт `"Me"`, так как там чёрным по белому написано, что мы изменяем значение переменной `name` на другое и возвращаем её текущее значение. Проблема в том, а видит ли `winner` это значение? Ведь мы изменяем значение той же переменной `name`, правда?

Буквальное понимание идеи замыкания приводит к мысли, что у каждого замыкания хранятся личные копии свободных переменных со значениями, которые они имели на момент создания замыкания. В таком случае предыдущая программа эквивалентна следующей:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (winner) )
```

В [Sam79] предлагается считать эффекты присваиваний видимыми только тем, кто это присваивание совершил. Тогда действительно `set-winner!` вернёт новое значение, а `winner` всегда будет возвращать `"Nemo"`. В таком мире не существует привязок. Нет, конечно у переменных есть значения, но они намертво установлены окружением. Если кому-то нужно значение переменной, то он может запросить его у окружения. Если же это значение надо изменить, то просто взамен старого окружения создаётся новое, где переменная имеет нужное значение.

Такая точка зрения напоминает способ реализации замыканий в старых, исключительно динамических диалектах Лиспа. Там замыкания создавались

явным использованием специальной формы `closure`; она принимала первым аргументом список переменных, которые надо сохранить, а вторым аргументом следовала функция. В результате вызов `(closure (x) (lambda (y) (+ x y)))` раскрывался в `(lambda (y) (let ((x '`*значение*`-x)) (+ x y)))`. С одной стороны, значение действительно захватывается, а с другой — всё укладывается в парадигму подстановки, унаследованную от λ-исчисления.

Но данная модель вычислений значительно усложняет как присваивание, так и совместное использование переменных. Возможны и другие варианты решения этой проблемы; в [SJ87] предлагается научить форму `closure` самостоятельно искать выделять свободные переменные в функции. Также, как предлагается в [BCSJ86, SJ93], она может определённым образом модифицировать найденные присваивания, чтобы избежать неудобств с присваиванием свободным переменным.

Scheme решает проблему иным путём, вводя понятие *привязок*, что также приводит к интересным *побочным эффектам*. Форма `let` привязывает к переменной `name` значение `"Nemo"`. Замыкания, создаваемые в теле `let`, захватывают не значение переменной `name`, а её привязку. Таким образом, ссылка на переменную `name` вызывает поиск соответствующей привязки с последующим извлечением значения из неё.

Присваивание действует аналогично: сначала ищется привязка, затем изменяется хранимое значение. В таком случае привязки являются объектами второго класса, которые не существуют отдельно от своих переменных. Присваивание переменной изменяет не саму привязку, а значение, на которое она указывает.

4.1.1. Коробки

Чтобы конкретизировать понятие привязок, давайте обратим внимание на A-списки. В предыдущих интерпретаторах A-списки использовались для представления окружений, их задачей было организовать для нас множество пар «переменная — значение». Они представляются точечными парами. Когда нам требуется переменная, мы ищем соответствующую точечную пару. В случае присваивания эта пара (точнее, её `cdr`) изменяет своё значение. Поэтому в данном случае именно точечные пары являются воплощением привязок. Но это не единственный способ их представления; например, привязки можно представлять с помощью *коробок* (boxes, также известны как cells). Новый уровень абстракции имеет большое значение, так как мы полностью избавляемся от присваивания переменным, заменяя его побочными эффектами операций над привязками.

Значение упаковывается в коробку с помощью функции `make-box`. Можно взглянуть на то, что в коробке, с помощью `box-ref` и положить в неё что-то другое с помощью `box-set!`. Набросаем реализацию коробок в стиле передачи сообщений:

```
(define (make-box value)
  (lambda (msg)
    (case msg
      ((get) value)
      ((set) (lambda (new-value) (set! value new-value))) ) ) )

(define (box-ref box)
  (box 'get) )

(define (box-set! box new-value)
  ((box 'set) new-value) )
```

Можно было бы реализовать их и без замыкания, используя точечные пары напрямую:

```
(define (other-make-box value)
  (cons 'box value) )

(define (other-box-ref box)
  (cdr box) )

(define (other-box-set! box new-value)
  (set-cdr! box new-value) )
```

Или вообще создать класс:

```
(define-class box Object (content))
```

Во всех трёх случаях (даже четырёх, если считать упражнение 3.10) мы явно указываем на неопределённость возвращаемого значения вызова `set-box!`. Каждую переменную, значение которой способно изменяться в процессе вычислений, можно представить как абстрактную коробку, сосредотачивая таким образом все побочные эффекты в реализации этих коробок. Заодно мы приобретаем возможность определить, изменяема ли переменная; для этого достаточно посмотреть на определяющую её связывающую форму и убедиться, что переменная упакована в коробку. (Это одна из заманчивых возможностей лексических окружений: чёткое определение области видимости переменных.)

$$
\begin{aligned}
\overline{x}^{v} &\Rightarrow \text{если } x = v, \text{ то } \texttt{(box-ref } v\texttt{)}, \text{ иначе } x \\
\overline{\texttt{(quote } \varepsilon\texttt{)}}^{v} &\Rightarrow \texttt{(quote } \varepsilon\texttt{)} \\
\overline{\texttt{(if } \pi_c\ \pi_t\ \pi_f\texttt{)}}^{v} &\Rightarrow \texttt{(if } \overline{\pi_c}^{v}\ \overline{\pi_t}^{v}\ \overline{\pi_f}^{v}\texttt{)} \\
\overline{\texttt{(begin } \pi_1\ \ldots\ \pi_n\texttt{)}}^{v} &\Rightarrow \texttt{(begin } \overline{\pi_1}^{v}\ \ldots\ \overline{\pi_n}^{v}\texttt{)} \\
\overline{\texttt{(set! } x\ \pi\texttt{)}}^{v} &\Rightarrow \text{если } x = v, \text{ то } \texttt{(box-set! } v\ \overline{\pi}^{v}\texttt{)}, \\
&\quad\ \text{иначе } \texttt{(set! } x\ \overline{\pi}^{v}\texttt{)} \\
\overline{\texttt{(lambda (...}\ x\ \texttt{...) } \pi\texttt{)}}^{v} &\Rightarrow \text{если } v \in \{\ldots\ x\ \ldots\}, \\
&\quad\ \text{то } \texttt{(lambda (...}\ x\ \texttt{...) } \pi\texttt{)}, \\
&\quad\ \text{иначе } \texttt{(lambda (...}\ x\ \texttt{...) } \overline{\pi}^{v}\texttt{)} \\
\overline{\texttt{(}\pi_0\ \pi_1\ \ldots\ \pi_n\texttt{)}}^{v} &\Rightarrow \texttt{(}\overline{\pi_0}^{v}\ \overline{\pi_1}^{v}\ \ldots\ \overline{\pi_n}^{v}\texttt{)}
\end{aligned}
$$

В приведённой выше таблице собраны правила перевода программ в «коробочный стиль». Запись $\overline{\pi}^v$ означает, что переменная v в программе π помещается в коробку. Как обычно, за именами переменных важно тщательно следить, чтобы не случилось коллизий.

Остаётся ввести ещё одно правило для собственно раскладывания по коробкам изменяемых переменных, чтобы полностью избавиться от присваиваний, заменив их побочными эффектами:

`(lambda (... `x` ...) `π`) ∧ (set! `x` ...) ∈ `π

→ `(lambda (... `x` ...) (let ((`x` (make-box `x`))) `$\overline{\pi}^x$`))`

В результате, предыдущий пример переписывается следующим образом:

```
(let ((name (make-box "Nemo")))
  (set! winner (lambda () (box-ref name)))
  (set! set-winner! (lambda (new-name) (box-set! name new-name)
                                       (box-ref name) ))
  (set-winner! "Me") (winner) )
```

Именно коробки обычно используются, когда в функцию надо передать изменяемую переменную; они фактически соответствуют ссылкам и указателям из иных языков. Коробки убирают из языка присваивания переменным и сопутствующие неоднозначности трактовки имени переменной: считать его ссылкой на переменную или же подставить вместо него значение. Благодаря коробкам не требуется введение специальных случаев при поиске значений переменных, так как *переменные* не могут быть изменены — если надо, то изменяется содержимое коробок, на которые эти переменные ссылаются. Конечно, изменение значения коробок требует побочных эффектов, так что ссылочная прозрачность всё так же остаётся утерянной. Тем не менее, это не мешает ввести систему типов содержимого коробок. С другой стороны, их использование вызывает свои специфичные проблемы.

Во-первых, привязки теперь становятся полноценными объектами, а значит, их можно передавать не только в `box-ref` и `box-set!`. Например, можно скопировать привязку и дать ей альтернативное имя; это может быть и полезным, например, при создании модулей.

Во-вторых, мы никак не можем контролировать использование коробок. Переменные теперь можно изменять в общем случае где угодно, передав копию коробки под другим именем. С лексическим присваиванием такой трюк не работает: для изменения значения переменной необходимо статически указать её правильное имя. Это, естественно, очень удобно для компилятора, так как он сразу же может понять, существует ли эта переменная вообще, является ли она свободной, известно ли о ней где-либо ещё; все это позволяет компилятору генерировать более эффективный код.

Также не стоит забывать, что в физическом компьютере каждая переменная связана со своим положением в памяти (адресом), где хранится её значение. При присваивании изменяется не сама переменная (не её адрес), а именно расположенное в памяти значение.

4.1.2. Присваивание свободным переменным

Следующая проблема с присваиванием касается свободных переменных: какой смысл оно имеет для них. Рассмотрим пример:

```
(let ((passwd "timhukiTrolrk"))  ; Это реальный пароль!
  (set! can-access? (lambda (pw) (string=? password (crypt pw)))) )
```

Переменная `can-access?` является свободной для этой формы. Более того, ей присваивается значение. По правилам Scheme `can-access?` должна быть глобальной, так как её нет в локальных связывающих формах. Но ведь то, что она *должна* быть где-то в глобальном окружении, вовсе не означает, что она там есть! Что делать в таком случае? Мы уже разговаривали на эту тему [**см. стр. 77**], рассмотрев при этом несколько вариантов поведения.

Что ещё можно делать с глобальной переменной, кроме присваивания? То же, что и с любой другой: ссылаться на неё, захватывать в замыканиях, получать её значение, определять перед использованием. Глобальное окружение переменных тоже является пространством имён со своими правилами, так что давайте рассмотрим подробнее различные варианты его реализации.

Всеобъемлющее глобальное окружение

Можно определить глобальное окружение как место, где все переменные предварительно объявлены. Естественно, в действительности они объявляются непосредственно перед первым использованием, но делается это незаметно и автоматически [Que95]. В таком мире с каждым именем связана одна и только одна переменная. Определение переменной здесь не имеет смысла, они все уже определены. Изменение переменной тоже не вызывает проблем: мы уверены, что изменяемая переменная существует. Соответственно, форма `define` фактически эквивалентна `set!` и, как следствие, можно безболезненно «определять» переменную несколько раз.

Единственная проблема возникает тогда, когда мы хотим получить значение переменной, но этой переменной никогда не присваивали значения. Переменная есть, но у неё нет значения. Часто результатом такой операции является сообщение о «неопределённой переменной», но в данном случае формально она определена, так что будет правильнее отвечать про «неинициализированную переменную».

Свойства данного окружения можно компактно представить следующей таблицей:

Ссылка	x
Значение	x
Изменение	`(set!` x `...)`
Расширение	запрещено
Определение	отсутствует, `define` $\equiv$ `set!`

Такой вариант окружения по-своему интересен: из-за того, что всё определено, уходят многие проблемы. Взаимно рекурсивные функции всегда смогут добраться друг до друга. Переопределить можно всё, что угодно. Но именно эти свойства являются источником других потенциальных проблем; с глобальными переменными надо быть осторожным, так как 1) они могут быть ещё не определены (но отобрать однажды данное значение уже нельзя); 2) их значение может меняться, а это значит, что на их текущее значение можно полагаться только в текущий момент. В частности, это отбирает у нас инлайн-функции, ведь в случае изменения значений переменных `car` или `cons` по идее придётся перекомпилировать всё, что их использовало до этого, что ставит под вопрос оправданность такого рода «оптимизаций».

Наконец, рассмотрим пример поведения переменных в данном окружении:

```
g                       ; ошибка: g не инициализирована
(define (P m) (* m g))
(define g 10)
(define g 9.81)         ; ≡ (set! g 9.81)
(P 10)        → 98.1
(set! e 2.78)           ; определение e
```

Короче говоря, такое глобальное окружение можно рассматривать как гигантскую форму `let`, определяющую все возможные переменные:

`(let (... a aa ... ab ... ac ...)` *программа*`...)`

Фиксированное глобальное окружение

Теперь представим, что с каждым именем связано не более одной глобальной переменной и перечень определённых переменных является неизменяемым. Такая ситуация возникает в скомпилированной программе без динамически генерируемого кода (без вызовов `eval`). Также, именно такое поведение было у всех наших предыдущих интерпретаторов: мы не можем создавать глобальные переменные в определяемом языке, все необходимые переменные создавались заранее с помощью формы `definitial` языка определения.

В таком окружении переменная существует только после явного создания с помощью `define`. Её значение можно считывать и изменять только после того, как переменная была определена. Тем не менее, существование не более одной переменной с уникальным именем всё же позволяет ссылаться на ещё не определённую переменную. (Это необходимое условие для взаимной рекурсии.) Естественно, больше одного раза определить глобальную переменную нельзя. В итоге, подобное окружение обладает следующими свойствами:

Ссылка	x
Значение	x, но x должна существовать
Изменение	`(set!` x `...)`, но x должна существовать
Расширение	`define` (единожды)
Определение	запрещено

Теперь попробуем запустить предыдущий пример в новом окружении:

```
g                       ; ошибка: неизвестная переменная g
(define (P m) (* m g)) ; опережающая ссылка на g
(define g 10)
(define g 9.81)         ; ошибка: переопределение g
(set! g 9.81)           ; изменение g
(P 10)          → 98.1
(set! e 2.78)           ; ошибка: неизвестная переменная e
```

После этого легко сообразить, как перенести обычные программы в такое окружение. Вся программа представляется последовательностью выражений $\pi_1 \dots \pi_n$. Она преобразуется в одно большое выражение следующим образом: проводится анализ выражений $\pi_1 \dots \pi_n$ для выделения всех свободных переменных; сами выражения помещаются в тело `let`-формы, которая определяет все свободные переменные как неинициализированные локальные переменные; наконец, все `define`-формы заменяются эквивалентными `set!`-формами.

Для пояснения рассмотрим следующую программу на Scheme:

```
(define (crypt pw) ...)

(let ((passwd "timhukiTrolrk"))
  (set! can-access? (lambda (pw) (string=? passwd (crypt pw)))) )

(define (gatekeeper)
  (until (can-access? (read))
         (gatekeeper) ) )
```

Эта небольшая программка спрашивает у пользователя пароль и не отпускает его, пока он не ответит правильно. Этот кибер-Цербер в новом окружении выглядит вот так:

```
(let (crypt make-can-access? can-access? gatekeeper)
  (set! crypt (lambda (pw) ...))
  (set! make-can-acceess?
        (lambda (passwd)
          (lambda (pw) (string?= passwd (crypt pw))) ) )
  (set! can-access? (make-can-access? "timhukiTrolrk"))
  (set! gatekeeper
        (lambda () (until (can-access? (read)) (gatekeeper))) )
  (gatekeeper) )
```

```
car ≡ car
cons ≡ cons
...
```

Глобальные определения заменяются локальными переменными, которые остаются неинициализированными до соответствующей `define`-формы.[1] Ко-

[1] К сожалению, непреднамеренно так вышло, что повторное определение переменной сейчас имеет смысл. Для точного соответствия необходимо будет внести коррективы в процедуру преобразования `define` в `set!` во избежание переопределения переменных.

нечно, встроенные функции вроде `read`, `string=?` или `string-append` всё так же остаются видимыми. В этом мире глобальное окружение конечно и ограничено предопределёнными функциями (вроде `car` и `cons`), а также свободными переменными наших программ. Присваивания глобальным переменным, не являющимся свободными, запрещены — свободных переменных, не присутствующих в глобальном окружении, быть не может по определению нашего преобразования. Единственный способ вызвать данную ошибку — это динамическое исполнение кода с помощью `eval`.

Автоматически расширяемое глобальное окружение

Если мы хотим создать REPL для своего языка, то нам необходима возможность динамического добавления переменных в глобальное окружение. Обычно это делается с помощью специальной формы `define`. Но можно сделать всё проще, используя лишь присваивание: если переменная не существует, то присваивание ей определяет её. В таком случае можно было бы писать просто:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (set! set-winner! (lambda (new-name) (set! name new-name)
                                       name )) )
```

В предыдущих случаях нам бы пришлось перед этим написать два бессмысленных утверждения вроде

```
(define winner      'жили-у-бабуси)
(define set-winner! 'два-весёлых-гуся)
```

Это бы определило две переменные и дало им начальные значения (потому что `define` не умеет определять переменные без значений), которые тут же изменяются на правильные соответствующими `set!`. Как видим, с таким окружением есть одна явная проблема: все переменные обязаны быть изменяемыми. В ранних версиях Scheme [SS78b] существовало специальное приспособление — форма `static`, — с помощью которой можно было[2] писать вот так:

```
(let ((name "Nemo"))
  (define (static winner) (lambda () name))
  (define (static set-winner!)
    (lambda (new-name) (set! name new-name)
                       name ) ) )
```

Таким образом можно было определять глобальные переменные, имея доступ к локальным, но при этом не пользуясь присваиваниями.

Хотя, несомненно, создавать глобальные переменные одним присваиванием удобно, мы рискуем нечаянно загрязнить глобальное окружение переменными, которые должны были быть локальными. Как вариант, можно разрешить создавать лишь локальные переменные; например, только лексические

[2] После того, как `define` объяснили, что `(static` *переменная*`)` — это ссылка на глобальную переменную, а не вызов унарной функции `static`.

с помощью `let`, или, как в [Nor72], только динамические переменные внутри `prog`.[3] К сожалению, это не вернёт ссылочную прозрачность, а кроме того, вызовет известные сложности с рекурсией.

Гиперстатическое глобальное окружение

Есть ещё один вариант глобального окружения: в нём с одним именем может быть связано несколько переменных, но новые переменные видны лишь тем формам, что следуют за их определением. Вернёмся к нашему любимому примеру:

```
g                       ; ошибка: неизвестная переменная g
(define (P m) (* m g)) ; ошибка: неизвестная переменная g
(define g 10)
(define (P m) (* m g))
(P 10)        → 100
(define g 9.81)
(P 10)        → 100     ; P использует старое значение g
(define (P m) (* m g))
(P 10)        → 98.1
(set! e 2.78)           ; ошибка: неизвестная переменная e
```

Здесь фраза «замыкание сохраняет свободные переменные из окружения определения» понимается буквально. При первом определении `P` в окружении нет переменной `g` — значит, это ошибка. Можно было бы допустить такое определение и вызывать ошибку при вызове `P`, но не стоит так делать, потому что `g` не динамическая переменная: если её не было при создании функции, то она не появится и при вызове. Незачем откладывать сообщение об ошибочном определении на потом.

Вторая версия `P` запоминает тот факт, что `g` равна 10 в окружении определения. В замыкании сохраняется именно этот факт: «свободная переменная `g` имеет значение `10`». Не важно, что случится с этой переменной в дальнейшем, для `P` она всегда равна десяти. Если мы хотим, чтобы `P` использовала новое значение, то `P` надо переопределить. Глобальное окружение является полностью лексическим; именно такое *гиперстатическое* окружение использует ML. Запишем его свойства в виде привычной таблицы:

Ссылка	x, но x должна существовать
Значение	x, но x должна существовать
Изменение	`(set!` x `...)`, но x должна существовать
Расширение	`define`
Определение	запрещено

[3] Например, так сделано в TEXe: определения, создаваемые `\def`, автоматически отменяются при выходе из текущей группы.

Но, как говорилось ранее, в таком случае возникают проблемы с рекурсивными определениями. Для Лиспа необходима возможность определения как простых рекурсивных функций, так и групп взаимно рекурсивных. В ML эта проблема решается ключевым словом `rec` для простой рекурсии и `and` для одновременных определений. В Scheme же есть `let` и `letrec` для аналогичных целей.

Наш пример в гиперстатическом окружении можно понимать следующим образом:

```
g                                   ; ошибка: неизвестная переменная g
(let ((P (lambda (m) (* m g))))     ; ошибка: неизвестная переменная g
  (let ((g 10))
    (let ((P (lambda (m) (* m g))))
      (P 10)
      (let ((g 9.81))
        ... ) ) ) )
```

Для полноты картины рассмотрим пример со взаимной рекурсией. Следующая программа на ML:

```
let rec odd n = if n = 0 then false else even (n - 1)
    and even n = if n = 0 then true else odd (n - 1)
```

легко переводится на Scheme следующим образом:

```
(letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
         (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
  ... )
```

Гиперстатическое окружение обладает очевидным преимуществом: возможностью выявлять неопределённые переменные статически. Кроме того, компилятору легко обнаружить фактически неизменяемые переменные, что позволяет ему применять некоторые оптимизации, основанные на статическом знании значения переменной. Тем не менее, у такого окружения есть и недостатки: например, в случае ошибки в определении приходится повторять все последующие определения, которые использовали исправленное ошибочное.

4.1.3. Присваивание предопределённым переменным

Кроме обычных свободных переменных программы на Scheme ссылаются также на предопределённые встроенные переменные вроде `car` и `read`. Естественно, у нас есть неотъемлемое право присваивать им значения, но какой смысл несут такие присваивания? Ведь часто интерпретатор опирается на неявные предположения о программах, чтобы выполнять их быстрее. Например, в Лиспе почти никогда не требуется переопределять `car` и `cdr`, поэтому они практически всегда реализуются как инлайн-функции и подставляются напрямую в код как своеобразные макросы. Переопределение любой из них сломает эту стройную систему, так как новые обращения к `car` должны будут

использовать текущее значение, а не встроенное; также неясно, что делать с предыдущими ссылками на подобные переменные, которые были (вероятно) заменены инлайн-кодом.

В гиперстатическом окружении последняя проблема, к счастью, отпадает: предыдущие ссылки будут видеть предыдущее значение. Но в динамическом глобальном окружении нам придётся отыскать все вхождения `car` и заменить их. Далее, по-видимому, потребуется заменить ещё и все вхождения `cadr`, которая является композицией `car` и `cdr`. В действительности, никто не занимается этим безнадёжным делом, так как это сложно, да и не похоже, что именно такой беспорядок ожидается пользователем в результате переопределения.

Между прочим, Scheme запрещает изменять значения глобальных примитивных функций. То есть создать свою глобальную переменную `car` нам-то ничто не помешает, но это никак не повлияет на `cadr`. Аналогично, функция `map` является встроенной, так что новая `car` не изменит её поведения, но, возможно, она затронет поведение какой-нибудь `mapc`, которая не входит в стандарт.

Часто глобальные примитивы изменяют для того, чтобы на скорую руку залатать ошибку в интерпретаторе, или, например, с целью подменить глобальную функцию её отладочным вариантом, который выводит на экран сообщения при каждом вызове. Тем не менее, я дам вам дружеский совет: «Ради бога, не трогайте примитивы, если у вас есть хоть капля сомнения в том, что именно получится в результате».

Подводя итог, гиперстатическое окружение ведёт себя просто, понятно и логично, но для отладки более удобным оказывается динамическое.

4.2. Побочные эффекты

До сих пор мы представляли вычисления упорядоченной тройкой выражения, окружения и продолжения. Как бы красиво она не выглядела, но с её помощью нельзя выразить присваивания, ввод-вывод и многое другое. Все эти глобальные изменения состояния мира объединяются в понятие *побочных эффектов* и для компьютера в конечном итоге сводятся к определённым изменениям в его памяти, изменениям её состояния.

Рассмотрим на знакомых примерах, почему это так. Если мы читаем что-то из потока, то где-то там изменяется положение метки, которая указывает на последний считанный байт. (А это не что иное, как побочный эффект, который мы получаем вдобавок к считанным данным.) Если мы уже вывели что-то на экран, то компьютер успеет провести миллионы операций за десять миллисекунд до следующего обновления экрана; а стереть напечатанное принтером он вообще не в состоянии. Попросить пользователя ввести что-то с клавиатуры — это тоже необратимо. Как видим, побочные эффекты вездесущи. На них основаны регистры, используемые для вычислений; без них мы бы не смогли сохранять наши драгоценные программы. Нет, конечно, можно представить

себе идеальный мир, где компьютеры работают без побочных эффектов, но кому нужен компьютер-аутист, которому ни объяснить, что от него требуется, ни получить от него ответ. Благо, этот кошмар не застанет нас наяву.

Мы уже видели, что присваивания можно заменить использованием коробок, но сами коробки используют точечные пары. Вопрос: можно ли реализовать точечные пары без точечных пар? Ответ: да, но для этого нам понадобится присваивание! Их можно представить в виде замыкания, реагирующего на сообщения[4] `car`, `cdr`, `set-car!` и `set-cdr!`:

```
(define (kons a d)
  (lambda (msg)
    (case msg
      ((car) a)
      ((cdr) d)
      ((set-car!) (lambda (new) (set! a new)))
      ((set-cdr!) (lambda (new) (set! d new))) ) ) )

(define (kar pair)
  (pair 'car) )

(define (set-kdr! pair value)
  ((pair 'set-cdr!) value) )
```

И вновь определение оказывается не вполне точным, как справедливо замечено в [Fel90], потому что теперь мы не можем различать точечные пары и замыкания. При таком определении нельзя написать предикат `pair?`, который будет работать верно в любых условиях. Если не обращать внимания на эту проблему (которую можно считать платой за возможность создавать свои типы данных), то чётко видно, что присваивание и побочные эффекты тесно связаны и легко выражаются друг через друга. Поэтому мы вынуждены или запретить использование их обоих, или же признать их существование и научиться контролировать возможные последствия. Какую именно форму использовать: присваивание переменным или прямую модификацию памяти, — зависит от контекста и желаемых свойств.

4.2.1. Равенство

Одним из главных неприятных последствий существования побочных эффектов является усложнение сравнения объектов. Когда можно считать два объекта равными? По мнению Лейбница, объекты эквивалентны, если они взаимозаменимы. С точки зрения программирования они взаимозаменимы, если мы не можем их различить. Эквивалентность обладает свойством рефлексивности: любой объект эквивалентен самому себе, так как очевидно, что он может заменить самого себя. Рефлексивность можно считать слабой эквивалентностью, если понимать её буквально; ведь мы именно так отличаем целые

[4] Будем писать эти имена через `k`, чтобы не путать их с обычными.

числа: каждое из них эквивалентно только самому себе. Но всё усложняется, когда кроме самих объектов нам доступны ещё и преобразования между ними. Например, пусть у нас есть два совершенно разных вычисления, вроде `(* 2 2)` и `(+ 2 2)`, и мы интересуемся, одно и то же ли они, взаимозаменимы ли они.

Если объекты можно изменять, то два объекта различны, если изменение одного не влияет на другой. Заметьте, что мы говорим о *различии*, а не о равенстве; но всё равно это достаточно важные классы объектов для понятия равенства: изменяемые и неизменяемые.

Кронекер говорил, что целые числа созданы Богом, поэтому логично считать, что других чисел быть не может, и число 3 будет оставаться тем же самым числом 3 в любом выражении. Аналогично можно причислить к неизменяемым все другие атомарные объекты (то есть те, которые не имеют составляющих частей), так что к числам отправляются также знаки, булевы значения и пустой список. Все остальные объекты являются составными, а значит, не первозданными и, следовательно, не обязательно неизменными.

Составные объекты вроде списков, строк и векторов логично считать равными, если они состоят из равных частей. Именно такой смысл равенства олицетворяют всевозможные варианты[5] предиката `equal?`. К сожалению, если состав объектов можно изменять, то равенство их в один момент времени не гарантирует сохранение равенства в любой другой. Поэтому мы введём предикат `eq?`, проверяющий идентичность: два объекта равны в смысле `eq?`, если они — это один и тот же объект. Такую проверку можно очень эффективно реализовать, сравнив адреса этих объектов: если они указывают на одно и то же место в памяти, то очевидно, что это один и тот же объект.

Итак, сейчас у нас есть два предиката: `eq?` для проверки идентичности и `equal?` для проверки структурного равенства. Но они оба впадают в крайности, когда дело касается изменяемых объектов. Равенство тождественно идентичности для неизменяемых объектов — два равных неизменяемых объекта такими и останутся навсегда. Но изменяемые объекты тождественны (всегда равны) только тогда, когда они идентичны (то есть суть один и тот же объект). Именно поэтому разумно понимать эквивалентность как взаимозаменимость: объекты эквивалентны, если ни одна программа не сможет их различить.

Предикат `eq?` работает на самом низком уровне: он может сравнивать только адреса ячеек памяти[6] и непосредственные константы. Некоторые объекты могут представляться в памяти по-разному, но быть при этом эквивалентными,[7] поэтому выходит, что `eq?` не обладает критичной для отношения эквивалентности рефлексивностью.

[5] Например, `equalp` и `tree-equal` в Common Lisp.

[6] В случае распределённых вычислений `eq?` должен также уметь сравнивать переменные, расположенные на различных компьютерах. Так что низкоуровневость этого предиката ещё не означает молниеносность его работы.

[7] Например, в Scheme значение `(eq? 33 33)` не определено. Для правильного сравнения чисел существует `eqv?`.

Существует улучшенный вариант `eq?`, называемый `eqv?` в Scheme и `eql` в Common Lisp. В целом это тот же `eq?`, но он работает немного медленнее, чтобы точно убедиться в равенстве неизменяемых объектов. Например, при сравнении чисел `eqv?` проверит, что они равны, даже если они по-разному представляются в памяти: с учётом всех форматов, длинной арифметики и т. п.

Если мы критически рассмотрим наши предыдущие интерпретаторы, то заметим, как мало в них изменяемых объектов. Практически всё, за исключением привязок, неизменяемо. Большая часть и так немногочисленных точечных пар никогда не видела в лицо `set-car!` или `set-cdr!`. В некоторых диалектах ML, а также в Common Lisp, можно явно указывать неизменяемые поля объектов. Вот так, например, определяется неизменяемая точечная пара:

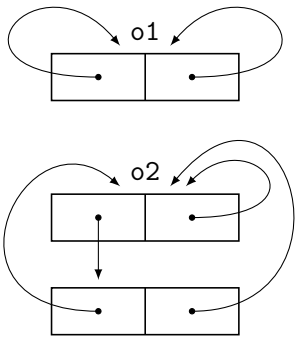
Common Lisp
```
(defstruct immutable-pair
  (car '() :read-only t)
  (cdr '() :read-only t) )
```

Такой способ, к примеру, удобен для представления цитированных точечных пар: их можно один раз создать, разместить в отдельной области памяти и, давая на них ссылку, можно быть уверенным, что их никто никогда не поменяет.

Генри Бейкер в [Bak93] предложил унифицировать предикаты равенства, объединив их в один — `egal`,* определяемый следующим образом: если сравниваемые объекты изменяемы, то он ведёт себя как `eq?`, в противном случае — как `equal?`. Истинность данного предиката гарантирует взаимозаменимость объектов. Это очень удобная функция, например, как показано в [QD93, Que94], для параллельного программирования.

Ещё одной проблемой являются циклические структуры данных. Их возможно сравнивать, но это обходится дорого. Если `equal?` не знает, как обращаться с такими структурами, то она просто застрянет в бесконечном цикле. Но даже если мы каким-то образом сможем определить наличие цикла, то всё ещё неясно, как проводить сравнение. Допустим, у нас есть вот такие объекты:

```
(define o1 (let ((pair (cons 1 2)))
             (set-car! pair pair)
             (set-cdr! pair pair)
             pair ))
(define o2 (let ((pair (cons (cons 1 2))))
             (set-car! (car pair) pair)
             (set-cdr! (car pair) pair)
             (set-cdr! pair pair)
             pair ))
```



*Немецкое слово egal и французское égal буквально значат «равен».

Предположим, мы хотим вычислить `(equal? o1 o1)`. Пусть `equal?` реализована так, что сначала вызывает `eq?`, и только потом — если ей переданы объекты, которые действительно могут быть не эквивалентны, — она выполняет долгую структурную проверку. В таком случае мы сразу же получаем ответ `#t`. Иначе `equal?` попадает в бесконечный цикл.[8]

Ладно, пусть результат `(equal? o1 o1)` зависит от реализации; что насчёт `(equal? o1 o2)`? И опять ответ зависит от того, изменяемы ли или нет точечные пары, из которых состоят `o1` и `o2`. Если они неизменяемы, то без `eq?` нельзя написать программу, которая бы смогла отличить `o1` от `o2`. Таким образом, лучше воздержаться от сравнения циклических структур данных с помощью `equal?`.

Рассмотренные предикаты поддерживают большую часть используемых типов данных, за исключением символов и функций. Символы часто являются не совсем атомарными, так как они могут содержать списки свойств. Но с точки зрения сравнения их можно считать такими же неизменяемыми, как и числа, потому что символ однозначно определяется своим именем; вся суть символов как раз в том, что не может быть двух различных символов с одинаковыми именами.

Что касается списков свойств, которые часто хранятся вместе с символами, то они являются удобным дополнением, например, для хранения всевозможных метаданных о глобальных переменных. В то же время, необходимо понимать ответственность за введение данной возможности: для её поддержки требуются побочные эффекты; свойства символа доступны глобально; для хранения каждого свойства требуются две точечные пары; желательно организовать эффективный поиск значения по имени свойства (например, на основе *хеш-таблиц*[9]).

4.2.2. Равенство функций

Случай сравнения функций может показаться безнадёжным. Легко догадаться, что функции эквивалентны, если они дают равные результаты для равных аргументов; в таком случае очевидно, что они взаимозаменимы. Более формально тождественность функций записывается так:

$$f \equiv g \iff \forall x\colon f(x) = g(x)$$

К сожалению, если бы всё было так просто запрограммировать… Мы ведь не можем всерьёз проверять все возможные значения аргументов. Поэтому придётся или отказаться от идеи сравнимости функций, или же принять менее

[8] Я запускал этот пример на четырёх различных интерпретаторах Scheme и два из них зациклились. Я не буду их называть, так как у них есть полное право так делать. [KCR98]

[9] По моему скромному мнению, изобретение хеш-таблиц является одним из величайших достижений информатики.

строгое определение эквивалентности для них. Сложно сказать, эквивалентны ли произвольные функции, но очень часто бывает легко выяснить, когда они не эквивалентны (например, если они принимают разное количество аргументов).

Поэтому для функций можно определить приближённое сравнение, понимая при этом, что оно может ошибаться, порой значительно. Естественно, пользоваться им можно только в том случае, если мы точно знаем, когда и насколько его результат будет ошибочным.

Scheme определяет `eqv?` для функций следующим образом: если есть хотя бы один набор аргументов, который приводит к различным результатам применения функций `f` и `g`, то данные функции не эквивалентны. И снова, здесь речь идёт о различии, а не о равенстве.

Давайте рассмотрим ещё несколько примеров. Выражение `(eqv? car cdr)` должно возвращать ложь, так как результаты применения этих функций значительно отличаются, например, для `(a . b)`.

Также очевидно, что `(eqv? car car)` должна возвращать истину, так как эквивалентность обязана быть рефлексивной. Но в некоторых реализациях данная форма возвращает ложь, так как `car` является инлайн-функцией и эта форма фактически читается как `(eqv? (lambda (x) (car x)) (lambda (x) (car x)))`. R^{5}RS оставляет результат `eqv?` при подобном сравнении на усмотрение реализации.

А как сравнить `cons` и `cons`, не прибегая к спасительной рефлексивности? Ведь функция `cons` создаёт изменяемые структуры данных, так что даже если её аргументы можно сравнить, то результаты вовсе не обязаны быть эквивалентными, так как они размещаются в разных участках памяти. Таким образом, `cons` не тождественна `cons`, и `(eqv? cons cons)` должна вернуть ложь, так как `(eqv? (cons 1 2) (cons 1 2))` возвращает ложь!

Вы всё ещё уверены, что `car` должна быть равна `car`? Есть и другие проблемы. Например, в языке без проверки типов `(car 'foo)` вызовет ошибку и выброс исключения, но результаты его перехвата и обработки слабо предсказуемы и зависят отнюдь не только от аргументов. Поэтому не все функции вообще могут вернуть что-то сравнимое. Это касается всех частичных функций,* которые используются вне своей фактической области определения.

Так что же делать? В различных языках применяются различные меры. Некоторые языки имеют мощную систему типов, поэтому они в состоянии определить, когда аргументы сравнения являются функциями; часть диалектов ML, например, этим пользуется, запрещая сравнивать функции вообще. В Scheme форма `lambda` создаёт замыкания. Каждый вызов `lambda` сохраняет в памяти замыкание, которое имеет определённый адрес, так что функции

* Функция $f\colon X \to Y$ называется частичной, если её значения на X могут быть не определены. Классический пример: операция вычитания для натуральных чисел имеет тип $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, но определена не для всех пар натуральных чисел $(a, b) \in \mathbb{N} \times \mathbb{N}$, только для тех, где $a \geqslant b$.

можно сравнивать хотя бы по этому адресу. Именно так и делает `eqv?`, считая функции с одинаковым адресом эквивалентными.

Однако, такое поведение может мешать некоторым оптимизациям. Рассмотрим функцию с говорящим именем `make-named-box`. Она принимает один аргумент (сообщение), анализирует его и реагирует соответствующим образом. Это один из возможных способов реализации объектов, именно его мы будем использовать для написания интерпретатора в этой главе.

```
(define (make-named-box name value)
  (lambda (msg)
    (case msg
      ((type) (lambda () 'named-box))
      ((name) (lambda () name))
      ((ref)  (lambda () value))
      ((set!) (lambda (new-value) (set! value new-value))) ) ) )
```

Замечательно, мы научились привязывать к коробкам бирку с именем. Теперь рассмотрим функцию внимательнее: замыкание, возвращаемое в ответ на сообщение `type`, не зависит от локальных переменных. Поэтому его можно вынести за скобки:

```
(define other-make-named-box
  (let ((type-closure (lambda () 'named-box)))
    (lambda (name value)
      (let ((name-closure  (lambda () name))
            (value-closure (lambda () value))
            (set!-closure  (lambda (new-value)
                             (set! value new-value) )) )
        (lambda (msg)
          (case msg
            ((type) type-closure)
            ((name) name-closure)
            ((ref)  value-closure)
            ((set!) set!-closure) ) ) ) ) ) )
```

Отлично, `type-closure` теперь создаётся только один раз для всех коробок, а не при каждом разборе `msg`.

Но что мы получим в результате следующего сравнения?

```
(let ((box (make-named-box 'foo 33)))
  (равны? (box 'type) (box 'type)) )
```

Это не совсем корректный вопрос, потому что здесь не указан предикат. Если это `eq?`, то ответ зависит от количества созданных копий `(lambda () 'named-box)`, если же это `egal`, то расположение замыканий не играет роли и ответом будет истина. Таким образом, семантика языка неявно зависит от предоставляемых им предикатов сравнения.

Если `lambda` создаёт замыкания в памяти, то у них всегда есть какой-то адрес. Два замыкания с одинаковым адресом, очевидно, являются одним и

тем же замыканием, а потому эквивалентны. Соответственно, в Scheme следующая программа ведёт себя верно:

```
(let ((f (lambda (x y) (cons x y))))
  (eqv? f f) ) → #t
```

Так как `eqv?` передан один и тот же объект, она вправе вернуть истину; даже если `(eqv? (f 1 2) (f 1 2))` возвращает ложь.

Как мы видим, при сравнении функций возникает множество вопросов, на которые нельзя дать однозначный ответ. Предпочтения зависят от свойств эквивалентности, которые желательно сохранить в языке, так как проверка математической тождественности функций не представляется возможной. Свойства подобного сравнения зависят от выбора авторов реализации языка. Иногда оно и вообще оказывается под запретом. Хотя в этой книге сравнение функций всё же используется для методов MEROONET [**см. стр. 530**], пожалуйста, если это возможно, старайтесь воздержаться от сравнения функций в любом виде.

4.3. Реализация

В этот раз мы напишем интерпретатор, основанный на замыканиях. Все его объекты будут `lambda`-формами, которые отправляют сообщения другим формам и разбирают входящие сообщения с помощью уже знакомой вам идиомы `(lambda (msg) (case msg ...))`. Некоторые сообщения будут универсальными для всех объектов. Сообщение `boolify` запрашивает у объекта его булев эквивалент (то есть `#t` или `#f` для нужд `if`-форм), а сообщение `type`, конечно же, вернёт его тип.

Главной проблемой является способ реализации побочных эффектов. Формально, переменная ссылается на привязку, которая указывает на значение этой переменной. Говоря простым языком, переменная указывает на коробку (адрес), где лежит её значение. Память — это функция, достающая из нужной коробки значение. Окружение — это функция, достающая из кучи нужную коробку. Всё это просто и понятно, но ровно до того момента, как мы попытаемся это запрограммировать.

Память должна быть видна отовсюду. Можно было бы сделать её глобальной переменной, но это не совсем элегантное решение, ведь вы знаете, сколько проблем и неоднозначностей таит в себе глобальное окружение. Есть и другой способ обеспечить видимость памяти для всех функций интерпретатора: передавать её как аргумент в каждую функцию, а она будет передавать её в следующую (возможно, изменив перед этим). Так и поступим; теперь вычисление описывается четвёркой из *выражения*, *окружения*, *продолжения* и *памяти*. В программе будем их коротко называть `e`, `r`, `k` и `s`.

Как обычно, функция `evaluate` проводит синтаксический анализ своего аргумента и вызывает соответствующую функцию-вычислитель. Не будем

нарушать установленную в предыдущих главах традицию именования сущностей, лишь расширим список новыми типами:

e, et, ec, ef	выражения, формы
r	окружения
k, kk	продолжения
v, void	значения
f	функции
n	идентификаторы
s, ss, sss	память
a, aa	адреса (коробки)

Так как у нас увеличилось количество аргументов функций, то отныне будем всегда их перечислять в определённом порядке: `e`, `r`, `s`, затем `k`.

Итак, ядро интерпретатора:

```
(define (evaluate e r s k)
  (if (atom? e)
      (if (symbol? e) (evaluate-variable e r s k)
          (evaluate-quote e r s k) )
      (case (car e)
        ((quote)  (evaluate-quote (cadr e) r s k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r s k))
        ((begin)  (evaluate-begin (cdr e) r s k))
        ((set!)   (evaluate-set! (cadr e) (caddr e) r s k))
        ((lambda) (evaluate-lambda (cadr e) (cddr e) r s k))
        (else     (evaluate-application (car e) (cdr e) r s k)) ) ) )
```

4.3.1. Ветвление

Условный оператор требует вспомогательного продолжения для выполнения ветвления после вычисления условия.

```
(define (evaluate-if ec et ef r s k)
  (evaluate ec r s
    (lambda (v ss)
      (evaluate ((v 'boolify) et ef) r ss k) ) ) )
```

Вспомогательное продолжение принимает не только вычисленное условие `v`, но и состояние памяти после вычисления. Следовательно, мы допускаем побочные эффекты при вычислении условия:

```
(if (begin (set-car! pair 'foo)
           (cdr pair) )
    (car pair) 2 ) → 'foo или 2
```

Сейчас любые изменения памяти, выполненные в условии, видны в обеих ветках. Но можно поступить иначе:

```
(define (evaluate-amnesic-if ec et ef r s k)
  (evaluate ec r s
    (lambda (v ss)
      (evaluate ((v 'boolify) et ef) r s     ; s ≠ ss!
                k) ) ) )
```

В этом случае при вычислении веток мы пользуемся старым состоянием памяти, которое было при вызове самой формы `if`. Это необходимое свойство для языков со встроенной поддержкой поиска с возвратом вроде Пролога. [**см. упр. 4.4**]

Так как мы решили все объекты интерпретатора представлять функциями, то в качестве логических значений нельзя использовать `#t` и `#f` языка реализации. Кроме этого нам необходим способ приведения любых объектов к логическому типу (ведь любой объект в Scheme, кроме `#f`, считается логической истиной). Будем считать, что все объекты отвечают на сообщение `boolify`, возвращая своё логическое представление в виде одного из комбинаторов λ-исчисления: `(lambda (x y) x)` или `(lambda (x y) y)`.

4.3.2. Последовательность

Хотя тут нет ничего интересного или нового, но повторить смысл последовательных вычислений будет не лишним. Для простоты мы считаем, что `begin` передана как минимум одна форма.

```
(define (evaluate-begin e* r s k)
  (if (pair? (cdr e*))
      (evaluate (car e*) r s
        (lambda (void ss)
          (evaluate-begin (cdr e*) r ss k) ) )
      (evaluate (car e*) r s k) ) )
```

Здесь чётко видно, как мы последовательно вычисляем тело `begin`, передаём новое состояние памяти, но игнорируем значения всех вычисляемых выражений, кроме последнего, которое обрабатывается отдельно.

4.3.3. Окружения

Окружения представляются функциями, переводящими имена переменных в адреса. Аналогично, память — это функция, переводящая адреса в значения. Начальное окружение должно быть пустым:

```
(define (r.init id)
  (wrong "No binding for" id) )
```

Итак, как же нам выразить изменение окружения (или памяти), не используя побочные эффекты или присваивание? В нашем случае это значит, что функция-память должна после изменения по-другому реагировать на опреде-

лённое значение, но по-старому — на все остальные. Функция `update` создаёт новую память именно с таким поведением:

```
(define (update s a v)
  (lambda (aa)
    (if (eqv? a aa) v (s aa)) ) )
```

Здесь буквально записана эта идея. Вызов `(update s a v)` вернёт новое, изменённое состояние памяти, которое мы передадим следующей функции как текущее. Также можно обобщить `update` для изменения сразу нескольких значений:

```
(define (update* s a* v*)
  ;; (assume (= (length a*) (length v*)))
  (if (pair? a*)
      (update* (update s (car a*) (car v*)) (cdr a*) (cdr v*))
      s ) )
```

Функция `update` одинаково хорошо работает как с памятью, так и с окружениями. В ML это была бы полиморфная функция, так как окружения и память имеют разные типы. Также необходимо учитывать, что `update` требуется некий критерий сравнения адресов, поэтому для их представления стоит выбрать гарантированно сравнимые объекты: например, целые числа. Имена переменных мы будем представлять символами. Оба типа данных прекрасно поддерживаются `eqv?`, поэтому в `update` используется именно этот предикат.

4.3.4. Обращения к переменным

Благодаря выбранному представлению памяти значение переменной получается элементарно:

```
(define (evaluate-variable n r s k)
  (k (s (r n)) s) )
```

Сначала мы извлекаем адрес переменной `(r n)` из окружения, затем ищем соответствующее значение в памяти `s` и передаём его продолжению `k`. Так как чтение памяти не вызывает побочных эффектов, то продолжению память передаётся в неизменном виде.

4.3.5. Присваивание

Для присваивания необходимо промежуточное продолжение.

```
(define (evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k v (update ss (r n) v)) ) ) )
```

Новое значение переменной после вычисления передаётся продолжению, которое записывает его в память по адресу переменной. Дальше промежуточное продолжение передаёт обновлённое состояние памяти изначальному продолжению формы `set!`. Между прочим, возвращаемое значение этого варианта присваивания вполне определено: это текущее значение переменной после присваивания.

Принцип работы памяти без побочных эффектов и причины представления её как функции теперь должны быть окончательно понятны: фактически, память представляется историей изменений своего состояния. Мы никогда не изменяем записи в этом списке и не удаляем их — только добавляем новые, переопределяя таким образом функцию-память. Конечно, если сравнивать его с реальной памятью, данный вариант является жутко неэффективным и избыточным, но ведение истории изменений имеет свои плюсы. [**см. упр. 4.5**] По крайней мере, можно взять эту идею на вооружение хотя бы частично.

4.3.6. Аппликация

Перед применением функции необходимо вычислить все элементы формы вызова. Вычислять их можно, как известно, в произвольном порядке. Пусть это будет порядок слева направо.

```
(define (evaluate-application e e* r s k)
  (define (evaluate-arguments e* r s k)
    (if (pair? e*)
        (evaluate (car e*) r s
          (lambda (v ss)
            (evaluate-arguments (cdr e*) r ss
              (lambda (v* sss)
                (k (cons v v*) sss) ) ) ) )
        (k '() s) ) )
  (evaluate e r s
    (lambda (f ss)
      (evaluate-arguments e* r ss
        (lambda (v* sss)
          (if (eq? (f 'type) 'function)
              ((f 'behavior) v* sss k)
              (wrong "Not a function" f) ) ) ) ) ) )
```

Здесь функция, которая обычно называлась `evlis`, является локальной и называется `evaluate-arguments`. Она вычисляет аргументы слева направо и собирает их в список. Затем функция (первый элемент формы) применяется к аргументам вместе с памятью после их вычисления и продолжением вызова функции. Обратите внимание на то, как лаконично и точно записаны эти идеи.

Функции представляются специальными замыканиями, так что напрямую их применять нельзя. Поэтому вводится сообщение `behavior` для извлечения поведения функции: кода, который будет применён к аргументам.

4.3.7. Абстракция

Для простоты будем считать, что специальная форма `lambda` поддерживает только функции с фиксированным количеством аргументов. При создании функции происходит две вещи: в памяти выделяется ячейка, затем создаётся особый объект — замыкание, — которое и размещается в только что выделенной ячейке памяти. Очевидно, что созданную функцию надо где-то хранить для дальнейшего использования, поэтому форма `lambda` вынуждена изменять состояние памяти. Работа по созданию и сохранению замыканий перекладывается на функцию `create-function`, которой передаётся адрес выделенного участка памяти и код функции — именно тот, который должно вернуть замыкание в ответ на сообщение `behavior`.

```
(define (evaluate-lambda n* e* r s k)
  (allocate 1 s
    (lambda (a* ss)
      (k (create-function
          (car a*)
          (lambda (v* s k)
            (if (= (length n*) (length v*))
                (allocate (length n*) s
                  (lambda (a* ss)
                    (evaluate-begin e*
                                    (update* r  n* a*)
                                    (update* ss a* v*)
                                    k ) ) )
                (wrong "Incorrect arity") ) ) )
          ss ) ) ) )
```

При вызове функции, созданной `lambda`, сначала проверяется количество переданных аргументов, затем в памяти в момент вызова выделяется место для аргументов функции, после чего они инициализируются переданными значениями и, наконец, управление передаётся собственно функции.

Возможен и другой вариант реализации:

```
(define (evaluate-ftn-lambda n* e* r s k)
  (allocate (+ 1 (length n*)) s
    (lambda (a* ss)
      (k (create-function
          (car a*)
          (lambda (v* s k)
            (if (= (length n*) (length v*))
                (evaluate-begin e*
                                (update* r n* (cdr a*))
                                (update* s (cdr a*) v*)
                                k )
                (wrong "Incorrect arity") ) ) )
       ss ) ) ) )
```

В этом случае место в памяти под аргументы выделяется один раз при создании замыкания; так поступает, например, Фортран. Но, к сожалению, такой подход лишает нас рекурсии, ведь теперь нет возможности хранить несколько отдельных наборов аргументов для каждого вызова функции. В Фортране подобное решение принято по той причине, что раз там нет динамического создания функций, то память для аргументов можно выделить ещё при компиляции, что приводит к росту производительности. Правда, ценой рекурсии и замыканий — главных достоинств функциональных языков.

4.3.8. Память

Память представляется функцией, принимающей адреса и возвращающей значения. Кроме этого нам нужна возможность выделять в памяти новые адреса, гарантированно свободные для использования; этим будет заниматься функция `allocate`. Она принимает три аргумента: необходимое количество адресов, память и продолжение, которому она передаст список выделенных адресов и новое состояние памяти, где по соответствующим адресам хранятся значения «не инициализировано». Именно на эту функцию возложена вся работа по управлению памятью, в частности, сборка мусора. Это отдельная сложная тема,* которой мы не будем здесь касаться и предположим, что память у нас бесконечная — то есть мы всегда можем найти необходимое количество свободных адресов.

```
(define (allocate n s k)
  (if (> n 0)
      (let ((a (new-location s)))
        (allocate (- n 1)
                  (expand-store a s)
                  (lambda (a* ss)
                    (k (cons a a*) ss) ) ) )
      (k '() s) ) )
```

Функция `new-location` ищет первый свободный адрес в памяти. Это настоящая чистая функция в том смысле, что `(eqv? (new-location s) (new-location s))` всегда возвращает истину. Если выделять ячейки в памяти последовательно, то достаточно будет хранить последний выделенный адрес для определения следующего свободного. Резервируется ячейка памяти с помощью функции `expand-store`.

```
(define (expand-store high-location s)
  (update s 0 high-location) )

(define (new-location s)
  (+ 1 (s 0) ) )
```

*Достаточно неплохое введение в сборку мусора есть в первом томе «Искусства программирования» Дональда Кнута.

Как видим, адрес последней выделенной ячейки сам хранится в памяти по адресу 0. Память отзывается на два типа сообщений: запрос адреса свободной ячейки (при передаче нуля) и запрос значения по адресу (при передаче иных адресов). Это просто и удобно для реализации. Естественно, также необходимо определить изначальное состояние памяти, где не выделено ни одной ячейки:

```
(define s.init
  (expand-store 0 (lambda (a) (wrong "No such address" a))) )
```

4.3.9. Представление значений

Мы решили представлять данные внутри интерпретатора в виде замыканий, обменивающихся сообщениями. Для простоты ограничимся пустым списком, булевыми значениями, символами, числами, точечными парами и функциями. Рассмотрим эти типы данных по очереди.

Все значения будут основываться на функции, которая отвечает как минимум на два сообщения: 1) запрос типа (`type`); 2) приведение к логическому типу (`boolify`). Значения конкретных типов данных будут реагировать на свои специфичные сообщения. Таким образом, все определения значений имеют следующий скелет:

```
(lambda (msg)
  (case msg
    ((type)    ...)
    ((boolify) ...)
    ... ) )
```

Существует только один пустой список и, в соответствии с R^{5}RS, для `if` он является истиной.

```
(define the-empty-list
  (lambda (msg)
    (case msg
      ((type)    'null)
      ((boolify) (lambda (x y) x)) ) ) )
```

Булевы значения создаются функцией `create-boolean`:

```
(define (create-boolean value)
  (let ((combinator (if value (lambda (x y) x)
                              (lambda (x y) y) )))
    (lambda (msg)
      (case msg
        ((type)    'boolean)
        ((boolify) combinator) ) ) ) )
```

Символы имеют имя, которое они назовут, если их вежливо попросить с помощью `name`. Мы будем представлять их имена родными символами Scheme.

```
(define (create-symbol v)
  (lambda (msg)
    (case msg
      ((type)    'symbol)
      ((name)    v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Числа же будут отвечать на сообщение `value`:

```
(define (create-number v)
  (lambda (msg)
    (case msg
      ((type)    'number)
      ((value)    v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Функции имеют более развитый словарный запас из двух сообщений: `tag` и `behavior`. Сообщение `behavior`, конечно же, вернёт код функции; `tag` хранит адрес, по которому эта функция расположена в памяти.

```
(define (create-function tag behavior)
  (lambda (msg)
    (case msg
      ((type)     'function)
      ((boolify)  (lambda (x y) x))
      ((tag)      tag)
      ((behavior) behavior) ) ) )
```

Остались только точечные пары. Точечная пара — это объект, состоящий из двух значений, каждое из которых можно изменять независимо. Поэтому мы будем их представлять в виде двух адресов: один будет ссылаться на `car`, второй — на `cdr`. Это может показаться необычным, так как традиционно точечные пары представляются не двумя отдельными коробками, а одной коробкой с двумя отделениями.[10] Это бы позволило чуть быстрее обращаться к её компонентам, а также применить известную технику размещения точечных пар в параллельных массивах: хранить `car` и `cdr` пар в двух отдельных массивах, но по одинаковым индексам; тогда адресом пары будет именно этот индекс. Но мы останемся верны идее привязок и будем хранить пары в виде двух адресов.

Для удобства мы также введём функцию, выделяющую в памяти целые списки. Она принимает список значений, которые необходимо разместить в памяти, собственно память и продолжение, которому эта функция передаст созданный список и новое состояние памяти.

[10] Причём по результатам серьёзных исследований вроде [Cla79, CG77], желательно, чтобы значение `cdr` (а не `car`), было доступно непосредственно. Другим преимуществом такого расположения может быть то, что точечные пары используются для представления связных списков, поэтому можно ввести общий класс «связных объектов» с единственным полем `cdr`.

```
(define (allocate-list v* s k)
  (define (consify v* k)
    (if (pair? v*)
        (consify (cdr v*) (lambda (v ss)
                            (allocate-pair (car v*) v ss k) ))
        (k the-empty-list s) ) )
  (consify v* k) )

(define (allocate-pair a d s k)
  (allocate 2 s
   (lambda (a* ss)
     (k (create-pair (car a*) (cadr a*))
        (update (update ss (car a*) a) (cadr a*) d) ) ) ) )

(define (create-pair a d)
  (lambda (msg)
    (case msg
      ((type)    'pair)
      ((boolify) (lambda (x y) x))
      ((set-car) (lambda (s v) (update s a v)))
      ((set-cdr) (lambda (s v) (update s d v)))
      ((car)     a)
      ((cdr)     d) ) ) )
```

4.3.10. Сравнение с объектным подходом

Замыкания, реагирующие на сообщения, с помощью которых мы представляем значения внутри интерпретатора, во многом напоминают объекты из предыдущей главы. Тем не менее, между подобными объектами-замыканиями и объектами MEROONET есть ряд отличий. Объекты-замыкания хранят поддерживаемые методы внутри; нет возможности добавлять однажды созданным объектам новые методы. Понятия классов и подклассов здесь присутствуют лишь умозрительно. В то же время, обобщённые функции позволяют расширять поведение объектов в любое время и в любом месте. С помощью обобщённых функций можно реализовать как методы, так и мультиметоды. Объекты из предыдущей главы поддерживают наследование, что позволяет выносить общие свойства в базовые классы, избегая дублирования кода. Поэтому не стоит думать об объектах как о «замыканиях для бедных», как считают некоторые фанатики Scheme, ведь даже объекты из предыдущей главы оказались значительно выразительнее. Хотя это не значит, что замыкания ни на что не способны [AR88].

4.3.11. Начальное окружение

Как обычно, мы определим два вспомогательных макроса, которые помогут создать начальное окружение (а также состояние памяти). Под каждую глобальную переменную необходимо выделить ячейку памяти, адрес которой будет храниться в глобальном окружении. Макрос `definitial` выделяет память и кладёт туда указанное значение.

```
(define s.global s.init)
(define r.global r.init)

(define-syntax definitial
  (syntax-rules ()
   ((definitial name value)
    (allocate 1 s.global
     (lambda (a* ss)
       (set! r.global (update r.global 'name (car a*)))
       (set! s.global (update ss (car a*) value)) ) ) ) ) )
```

Макрос `defprimitive` определяет глобальные функции.

```
(define-syntax defprimitive
  (syntax-rules ()
   ((defprimitive name value arity)
    (definitial name
     (allocate 1
      s.global
      (lambda (a* ss)
        (set! s.global (expand-store (car a*) ss))
        (create-function (car a*)
         (lambda (v* s k)
           (if (= arity (length v*))
               (value v* s k)
               (wrong "Incorrect arity" 'name ) )) ) ) ) ) ) ) )
```

По уже сложившейся традиции, определим глобальные переменные для истины, лжи и пустого списка:

```
(definitial t (create-boolean #t))
(definitial f (create-boolean #f))
(definitial nil the-empty-list)
```

В качестве примера определения примитивов покажем предикат и арифметическую операцию. Они проверяют типы аргументов, извлекают значения, выполняют соответствующие действия и упаковывают результат во внутреннее представление.

```
(defprimitive <=
  (lambda (v* s k)
    (if (and (eq? ((car v*)  'type) 'number)
             (eq? ((cadr v*) 'type) 'number) )
```

```
        (k (create-boolean (<= ((car v*) 'value)
                               ((cadr v*) 'value) )) s)
        (wrong "Numbers required" '<=) ) )
  2 )

(defprimitive *
  (lambda (v* s k)
    (if (and (eq? ((car v*)  'type) 'number)
             (eq? ((cadr v*) 'type) 'number) )
        (k (create-number (* ((car v*) 'value)
                             ((cadr v*) 'value) )) s)
        (wrong "Numbers required" '*) ) )
  2 )
```

4.3.12. Точечные пары

Впервые точечные пары внутри нашего интерпретатора Scheme не представляются родными точечными парами Scheme, на котором он написан.

Так как у нас есть функция `allocate-pair`, то определить `cons` легко:

```
(defprimitive cons
  (lambda (v* s k)
    (allocate-pair (car v*) (cadr v*) s k) )
  2 )
```

С чтением и модификацией тоже нет проблем:

```
(defprimitive car
  (lambda (v* s k)
    (if (eq? ((car v*) 'type) 'pair)
        (k (s ((car v*) 'car)) s)
        (wrong "Not a pair" (car v*)) ) )
  1 )

(defprimitive set-cdr!
  (lambda (v* s k)
    (if (eq? ((car v*) 'type) 'pair)
        (let ((pair (car v*)))
          (k pair ((pair 'set-cdr) s (cadr v*))) )
        (wrong "Not a pair" (car v*)) ) )
  2 )
```

Все значения представляются таким образом, что узнать их тип не составляет труда — достаточно передать объекту сообщение `type`. Поэтому написать предикат `pair?` тоже просто, надо только не забыть перевести результат сравнения во внутреннее представление.

```
(defprimitive pair?
  (lambda (v* s k)
    (k (create-boolean (eq? ((car v*) 'type) 'pair)) s) )
  1 )
```

4.3.13. Функция сравнения

Одной из задач данного интерпретатора является иллюстрация реализации предиката эквивалентности — `eqv?`. Он проверяет, являются ли два объекта взаимозаменимыми. Сначала он сравнивает типы объектов, затем, если они совпадают, он проводит собственно сравнение. Символы должны иметь одинаковое имя. Логические значения и числа должны быть одинаковыми. Точечные пары и функции должны иметь одинаковые адреса.

```
(defprimitive eqv?
  (lambda (v* s k)
    (k (create-boolean
        (if (eq? ((car v*) 'type) ((cadr v*) 'type))
            (case ((car v*) 'type)
              ((null) #t)
              ((boolean)
               (((car v*) 'boolify)
                (((cadr v*) 'boolify) #t #f)
                (((cadr v*) 'boolify) #f #t) ) )
              ((pair)
               (and (= ((car v*) 'car) ((cadr v*) 'car))
                    (= ((car v*) 'cdr) ((cadr v*) 'cdr)) ) )
              ((symbol)
               (eq? ((car v*) 'name) ((cadr v*) 'name)) )
              ((number)
               (= ((car v*) 'value) ((cadr v*) 'value)) )
              ((function)
               (= ((car v*) 'tag) ((cadr v*) 'tag)) )
              (else #f) )
            #f ) )
       s ) )
  2 )
```

Собственно, единственная причина, по которой мы сохраняем адреса функций (а точнее, замыканий) внутри объектов — это сравнение функций с помощью `eqv?`. Таким образом, вместо решения сложной проблемы эквивалентности функций достаточно ответить на простой вопрос о равенстве адресов, которые суть просто числа. Заметьте, что при сравнении точечных пар мы тоже сравниваем лишь адреса компонент, а не их содержимое. Итого, `eqv?` выполняется за предсказуемое постоянное время, в отличие от `equal?`.

Функция `eqv?` похожа на обобщённую, только она содержит все методы внутри, а в остальном — всё та же диспетчеризация по типу аргумента. Благодаря выбранному представлению данных мы можем быстро провести проверку на согласованность типов и перейти непосредственно к сравнению значений или адресов, которое чаще всего тоже выполняется быстро (длинной арифметике не повезло).

4.3.14. Запускаем интерпретатор

Наконец, всё готово для запуска нашего интерпретатора. Главный цикл вызывает `evaluate` с рекурсивным продолжением, которое обеспечивает непрерывность сознания интерпретатора, передавая следующему вызову память предыдущего.

```
(define (chapter4-interpreter)
  (define (toplevel s)
    (evaluate (read) r.global s
              (lambda (v ss)
                (display (transcode-back v ss))
                (toplevel ss) ) ) )
  (toplevel s.global) )
```

Главной особенностью данного интерпретатора является отличие представления данных интерпретируемого языка от представления данных в языке реализации. Особенно разительно отличаются точечные пары и, следовательно, списки, поэтому мы вынуждены воспользоваться функцией `transcode-back` для преобразования результата вычислений в нечто понятное `display`, чтобы его можно было вывести на экран. Эту вспомогательную функцию вполне можно сразу объединить с примитивом `display`.

```
(define (transcode-back v s)
  (case (v 'type)
    ((null)     '())
    ((boolean)  ((v 'boolify) #t #f))
    ((symbol)   (v 'name))
    ((number)   (v 'value))
    ((pair)     (cons (transcode-back (s (v 'car)) s)
                      (transcode-back (s (v 'cdr)) s) ))
    ((function) v)  ; почему бы и нет?
    (else       (wrong "Unknown type" (v 'type))) ) )
```

4.4. Ввод-вывод и память

Теперь поговорим о вводе-выводе. Если ограничиться одним потоком ввода и одним потоком вывода (то есть функциями `display` и `read`), то достаточно добавить каждой функции интерпретатора ещё два аргумента и передавать через них эти потоки. Поток ввода содержит всё, что программа может прочитать, а поток вывода содержит всё, что она туда написала. Можно даже предположить, что эти потоки являются единственным средством связи программы с внешним миром.

Поток вывода можно легко представить списком пар (*память*, *значение*), которые передаются функции `transcode-back`. Но как представить поток ввода? Хитрость здесь в том, что мы можем вводить и точечные пары со спис-

ками, а они не могут существовать отдельно от памяти. Поэтому очевидно, что в случае чтения точечных пар необходимо сохранять в памяти их компоненты. Переводчиком будет служить функция `transcode`, которая принимает значение языка реализации (обозначим его `c`), память и продолжение. Она переводит это значение в представление реализуемого языка и передаёт его продолжению вместе с новым состоянием памяти.

```
(define (transcode c s k)
  (cond
    ((null? c)    (k the-empty-list s))
    ((boolean? c) (k (create-boolean c) s))
    ((symbol? c)  (k (create-symbol c) s))
    ((number? c)  (k (create-number c) s))
    ((pair? c)
     (transcode (car c)
                s
                (lambda (a ss)
                  (transcode (cdr c)
                             ss
                             (lambda (d sss)
                               (allocate-pair a d sss k) ) ) ) ) )
    (else (wrong "Not supported" c)) ) )
```

На этом мы остановимся, так как дальнейшая реализация слишком объёмна, чтобы приводить её здесь полностью: помимо примитивов `read` и `display`, необходимо в каждой функции, которую мы написали до этого, расширить набор аргументов `e`, `r`, `s`, `k` ещё двумя: `i` и `o`, через которые и передавать потоки ввода-вывода подобно памяти (потому что они, как и память, должны быть доступны отовсюду).

4.5. Семантика цитирования

Наверное, вы уже заметили, что в этот раз мы как-то обошли стороной цитирование. Форма `quote` раньше всегда записывалась элементарно, но только потому, что представление значений в интерпретируемом языке совпадало с представлением значений в самом интерпретаторе. Теперь это не так, поэтому определять `quote` следующим образом нельзя:

```
(define (evaluate-quote v r s k)
  (k v s) )
```
Ошибка!

Здесь ошибочно считается, что `v` — это именно то значение, которое необходимо вернуть. В смысле, конечно, это именно то, что надо, но это значение записано на языке, которого наш интерпретатор не понимает. В предыдущих главах интерпретатор пользовался языком реализации, но теперь ему необходим переводчик — `transcode`. Поэтому (более-менее) правильное определение цитирования выглядит так:

```
(define (evaluate-quote c r s k)
  (transcode c s k) )
```

Такое определение цитирования *композиционально* — его смысл зависит только от аргумента `c`, оно не учитывает контекст (значения других переменных). Но, к сожалению, точный смысл таких цитат немного отличается от того, который обычно подразумевается в Лиспе и Scheme. Рассмотрим выражение `(quote (a . b))`. По определению оно *в точности* эквивалентно `(cons 'a 'b)`.[11] Когда мы цитируем составной объект вроде точечной пары `(a . b)`, мы создаём новую пару в памяти. Таким образом, такое цитирование — это лишь краткая запись для наших `create-`функций. Поэтому ничуть не удивительно, что следующее выражение возвращает ложь:

```
(let ((foo (lambda () '(f o o))))
  (eq? (foo) (foo)) )
```

Функция `foo` каждый раз создаёт новый список; все создаваемые значения независимы, различны в смысле `eq?`. Если мы хотим, чтобы данное выражение всегда возвращало истину, нам необходимо доработать `evaluate-quote` так, чтобы она возвращала идентичные значения, а не эквивалентные; для этого она должна запоминать всё, что возвращала ранее. Это называется *мемоизация*. Одной из причин желательности такого поведения является то, что если в языке нет побочных эффектов и `eq?`, то нельзя отличить идентичные объекты от просто эквивалентных, поэтому нет никакого смысла тратить память на бесполезные копии эквивалентных объектов, когда можно обойтись одним экземпляром.

Конечно, если допускать побочные эффекты, то дело меняется, но всё же обычно считается, что цитаты уникальны — как числа, символы и логические значения.

```
(define *shared-memo-quotations* '())
(define evaluate-memo-quote
  (lambda (c r s k)
    (let ((couple (assoc c *shared-memo-quotations*)))
      (if (pair? couple)
          (k (cdr couple) s)
          (transcode c s (lambda (v ss)
                           (set! *shared-memo-quotations*
                                 (cons (cons c v)
                                       *shared-memo-quotations* ) )
                           (k v ss) )) ) ) ) )
```

Однако, такой вариант, во-первых, требует побочных эффектов и глобальной переменной, чего хотелось бы избежать, а во-вторых, учёт используемых цитат вполне можно выполнить и заранее, за что компилятор скажет нам спасибо: сначала мы переносим вычисление цитат в самое начало программы,

[11] Ещё точнее: `'(a . b)` ≡ `` `(,'a . ,'b) ``.

затем заменяем все обращения к цитатам ссылками на переменные, хранящие вычисленные ранее значения. На предыдущем примере это выглядит так:

```
(let ((foo (lambda () '(f o o))))    (define quote35 '(f o o))
  (eq? (foo) (foo)) )             ⇝ (let ((foo (lambda () quote35)))
                                       (eq? (foo) (foo)) )
```

В итоге цитаты обретают должный смысл, а `evaluate-quote` остаётся такой же простой.

Можно было бы пойти и дальше, полностью избавившись от неоднозначностей, связанных с цитированием составных объектов, сделав их создание явным:

```
(define quote36 (cons 'o '())
(define quote37 (cons 'o quote36)
(define quote38 (cons 'f quote37)

(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

И ещё дальше, на всякий случай гарантируя правильное цитирование символов вручную:

```
(define symbol39 (string->symbol "o"))
(define symbol40 (string->symbol "f"))

(define quote36 (cons symbol39 '())
(define quote37 (cons symbol39 quote36)
(define quote38 (cons symbol40 quote37)

(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

Пожалуй, на этом остановимся. На языке ассемблера или Си строки можно представлять непосредственно, так что было бы глупо разбивать их на отдельные символы — а то так ведь можно и до элементарных частиц дойти!

В конце концов, композициональное определение цитирования оказывается приемлемым, так как имеет простую формулировку, которая хоть и отличается по смыслу от традиционного понимания цитирования, но это легко поправимо простым дополнительным преобразованием программ.

Однако, унифицированное представление кода и данных может сыграть с нами злую шутку. С одной стороны это удобно: читать и данные, и программы с помощью одной функции `read`. Для того, чтобы она могла отличить формы-данные от форм-программ, как раз и существует специальная форма `quote`. Но в том-то и дело, что после `quote` данные становятся частью программы. Рассмотрим пример:

```
(define vowel<=
  (let ((vowels '(#\a #\e #\i #\o #\u)))
    (lambda (c1 c2)
      (memq c2 (memq c1 vowels)) ) ) )
```

```
(set-cdr! (vowel<= #\a #\e) '())
(vowel<= #\o #\u) → ?
```

При интерпретации этой программы есть высокая вероятность получить не `#t`, а ошибку. И если после этого мы распечатаем определение `vowel<=` (или если бы `vowels` была глобальной), то увидим следующее:

```
(define vowel1<=
  (let ((vowels '(#\a #\e)))
    (lambda (c1 c2)
      (memq c2 (memq c1 vowels)) ) ) )
```

Ой, мы нечаянно кусок программы! Аналогично можно добавлять элементы в список `vowels` или изменять их. Правда, такие трюки можно проделывать только в интерпретаторе; в случае компилятора мы скорее всего получим ошибку. Дело в том, что если глобальная переменная `vowel<=` неизменяема, то все формы вроде `(vowel<= #\a #\e)` могут быть вычислены ещё на этапе компиляции, чем компилятор и пользуется, подставляя вместо них сразу готовый результат. Такая оптимизация называется *свёрткой констант* (constant folding), её обобщением является техника частичных вычислений [JGS93].

Компилятор также может решить подставить константы и в само определение функции `vowel<=` ради небольшого ускорения:

```
(define vowel2<=
  (lambda (c1 c2)
    (case c1
      ((#\a) (memq c2 '(#\a #\e #\i #\o #\u)))
      ((#\e) (memq c2 '(#\e #\i #\o #\u)))
      ((#\i) (memq c2 '(#\i #\o #\u)))
      ((#\o) (memq c2 '(#\o #\u)))
      ((#\u) (eq? c2 #\u))
      (else  #f) ) ) )
```

Теперь выражение `(eq? (cdr (vowel<= #\a #\e)) (vowel<= #\e #\i))` имеет неопределённое значение. С одной стороны, оно не должно быть истиной, так как сравниваемые значения принадлежат явно различным цитатам. С другой стороны, иногда оно всё же истинно: компиляторы часто оставляют за собой право оптимизировать расположение констант в памяти, накладывая их друг на друга и превращая исходную программу в следующую:

```
(define quote82 (cons #\u '()))
(define quote81 (cons #\o quote82))
(define quote80 (cons #\i quote81)
(define quote79 (cons #\e quote80)
(define quote78 (cons #\a quote79)

(define vowel3<=
  (lambda (c1 c2)
    (case c1
```

```
      ((#\a) (memq c2 quote78))
      ((#\e) (memq c2 quote79))
      ((#\i) (memq c2 quote80))
      ((#\o) (memq c2 quote81))
      ((#\u) (eq? c2 #\u))
      (else  #f) ) ) )
```

Такая оптимизация влияет на разделяемые цитаты и ломает `eq?`. Есть простой способ избежать всех этих проблем: запретить изменять цитаты. Именно так и поступают Scheme и Common Lisp. Но они делают это мягко, в общем случае считая результат изменения значения цитаты неопределённым.

Можно поступить более жёстко, запретив изменение цитат синтаксически:

```
(define (evaluate-immutable-quote c r s k)
  (immutable-transcode c s k) )

(define (immutable-transcode c s k)
  (cond
    ((null? c)    (k the-empty-list s))
    ((pair? c)
     (immutable-transcode
      (car c) s (lambda (a ss)
                  (immutable-transcode
                   (cdr c) ss (lambda (d sss)
                                (allocate-immutable-pair
                                 a d sss k ) ) ) ) ) )
    ((boolean? c) (k (create-boolean c) s))
    ((symbol? c)  (k (create-symbol c) s))
    ((number? c)  (k (create-number c) s))
    (else (wrong "Not supported" c)) ) )

(define (allocate-immutable-pair a d s k)
  (allocate 2 s
   (lambda (a* ss)
     (k (create-immutable-pair (car a*) (cadr a*))
        (update (update ss (car a*) a) (cadr a*) d) ) ) ) )

(define (create-immutable-pair a d)
  (lambda (msg)
    (case msg
      ((type)    'pair)
      ((boolify) (lambda (x y) x))
      ((set-car) (lambda (s v) (wrong "Immutable pair")))
      ((set-cdr) (lambda (s v) (wrong "Immutable pair")))
      ((car)     a)
      ((cdr)     d) ) ) )
```

В таком случае любая попытка модифицировать цитату обречена на провал. Подобным образом можно ввести неизменяемые строки (как это сделано, например, в Mesa), неизменяемые векторы и т. д.

Подводя итог, лучше воспользоваться мудростью поколений и, как советуют Scheme и Common Lisp, стараться не изменять значения цитат. Но это не последняя из проблем, вызываемых цитированием. Мы уже упоминали методику «склеивания» цитат, приводящую к физическому наложению данных. А можно ли намеренно создать цитату с таким эффектом? Есть как минимум два способа это сделать: макросы и программируемый поток ввода.

В Common Lisp можно вводить специальные макросимволы, при чтении которых вызываются определённые процедуры. Например, `'` является именно макросимволом, оборачивающим следующее прочитанное выражение в `quote`. Или `#.`*выражение*, вместо которого подставляется значение *выражения*; то есть следующее за `#.` выражение считывается, вычисляется[12] и его значение подставляется вместо всей конструкции. В итоге, мы можем написать так:

Common Lisp
```
(define bar (quote #.(let ((list '(0 1)))
                       (set-cdr! (cdr list) list)
                       list )))
```

После этого переменная `bar` будет содержать бесконечный список из чередующихся нулей и единиц. С помощью `#.` мы процитировали пару, чей `cddr` равен ей самой. В CLtL2 [Ste90] прямым текстом написано, что `#.` и существует для ввода таких неудобных значений. Если мы захотим реализовать такое поведение у себя, то это немного усложнит трансформацию программ (которая выносит вычисление цитат в начало), так как она должна будет учитывать возможность циклов.

Хорошая новость: в Scheme так писать нельзя, так как `read` не программируется и макросимволы не поддерживаются. Плохая новость: в Scheme есть просто макросы, с помощью которых можно добиться того же результата. Форма `define-syntax`, правда, такого не позволяет, поэтому мы используем `define-abbreviation` — макрос для определения макросов, который детально рассматривается в девятой главе. [**см. стр. 370**]

```
(define-abbreviation (cycle n)
  (let ((c (iota n)))           ; (iota 3) → (0 1 2 3)
    (set-cdr! (last-pair c) c)  ; (last-pair '(1 2 3)) → (3)
    `(quote ,c) ) )
(define bar (cycle 2))
```

Таким образом в Scheme определяется циклический список из нулей и единиц, подставляемый в цитату, которой инициализируется переменная `bar`. Мы не можем написать такой список прямым текстом в определении `bar`, так как

[12] Окружение и продолжение вычислений выбираются на усмотрение реализации.

`read` Scheme не умеет считывать циклические данные. В Common Lisp это возможно с помощью макросимволов `#`n`=` и `#`n`#`[13]:

```
(define bar #1=(0 1 . #1#))
```

В настоящее время в Scheme подобные списки нельзя записывать непосредственно в виде *литералов*, только создавать вручную за несколько шагов. Поэтому и цитировать подобные структуры можно только лишь с помощью ухищрений вроде макросов.

Девятая глава будет посвящена подробному рассмотрению макросов, их возможностей, ограничений и вызываемых ими проблем. [**см. стр. 370**]

4.6. Заключение

Данная глава содержит колоссальное количество советов и описанных подводных ям. Глобальные переменные таят множество неоднозначностей, которых становится ещё больше с появлением модулей. Универсального определения равенства нет даже в математике, что уж говорить о программировании. Наконец, цитирование также отнюдь не такое простое, каким кажется; от витиеватых цитат стоит воздерживаться.

4.7. Упражнения

Упражнение 4.1 Определите чистую (без побочных эффектов) функцию `min-max`.

Упражнение 4.2 Точечные пары, которые мы реализовали с помощью замыканий, реагируют на сообщения-символы. Запретите модификацию пар (`set-car!` и `set-cdr!`) и реализуйте их, используя исключительно `lambda`-формы.

Упражнение 4.3 Определите `eq?` для точечных пар с помощью `set-car!` или `set-cdr!`.

Упражнение 4.4 Определите новую специальную форму `(or` α β`)`, которая возвращает значение α, если оно приводится к истине; иначе `or` откатывает все побочные эффекты вычисления α и возвращает значение β.

Упражнение 4.5 Присваивание в текущем варианте возвращает только что присвоенное значение. Перепишите `set!` так, чтобы она возвращала значение переменной до присваивания.

[13] Здесь считываемый объект ссылается на самого себя. Следовательно, функция `read` должна уметь создать в памяти точечную пару, «зациклить» её и подставить адрес на место макроса. Ей также желательно быть достаточно сообразительной, чтобы не попасть впросак на чём-нибудь вроде `#1=#1#`.

Упражнение 4.6 Определите функции `apply` и `call/cc` для интерпретатора из этой главы.

Упражнение 4.7 Встройте в интерпретатор поддержку функций переменной арности (с точечным аргументом).

Глава 5

Денотационная семантика

Эта глава начинается кратким обзором λ-исчисления, после чего переходит к денотационной семантике во всей её красе. Мы рассмотрим ещё одно определение Лиспа — в этот раз денотационное, — значительно отличающееся от предыдущих интерпретаторов: смысл программ теперь будет выражаться математическими объектами — термами λ-исчисления.

Чем именно является программа? *Программа* — это описание вычислений, которые приводят к определённому результату: значению и/или эффекту.

Очень часто программу путают с её исполнимым воплощением для той или иной машины; аналогично, определением программы часто считают файл с её исходным кодом; но всё это совершенно различные понятия.

Программы записываются на некотором *языке*; определение этого языка придаёт корректным программам *смысл*. Смысл программы — это не только и не столько результаты вычислений, так как они зависят от введённых данных, от событий во внешнем мире, от действий других программ. Смысл программы — это нечто большее, это суть проводимых вычислений.

Если мы хотим как-то изучать смысл программ и обрабатывать их, то они обязаны иметь математическое представление. Возьмём любое преобразование, например, перевод программы в «коробочный стиль» из предыдущей главы. [**см. стр.** 146] Как мы можем быть уверены, что после подобной операции смысл программы останется неизменным? Только с помощью тысячелетнего опыта математики — науки об отношениях, структуре и преобразованиях. А для этого необходимо связать смысл программ с математическими объектами. Такой подход кажется вполне разумным и полезным: например, если бы функция `fact` выражалась буквально через факториал, то стало бы гораздо проще убедиться в том, что она вычисляет именно факториал, или что другие функции, вроде `meta-fact` [**см. стр.** 88], действительно ей эквивалентны.

Для определения смысла программы необходим её математический эквивалент, а для его построения нам надо в точности знать свойства языка, на котором написана программа. В общем, проблема сводится к отысканию способа построения математических эквивалентов всех конструкций языка. То есть к формализации его *семантики*. Семантика языка программирования даёт нам его полное понимание: мы можем реализовать язык на чём угодно, мы можем доказывать правильность программ и их преобразований, мы можем

сравнивать языки между собой и многое другое. Применений семантики великое множество, но не только она одна даёт такие возможности.

Одним из древнейших способов определения языков является предоставление *эталонной реализации*. Когда необходима некая информация о языке, например, последствия выполнения какой-либо программы, то эту программу можно передать эталонной реализации и получить ответ: каким должно быть возвращаемое значение или побочные эффекты. Но, как и с любым чёрным ящиком, очень непросто построить полную теорию его работы, пользуясь лишь подобными наблюдениями. Если мы решим открыть ящик, то внутри окажется ещё одна программа на каком-то языке, что возвращает нас к исходному вопросу о смысле программ.

Следующий подход основывается на идее *виртуальной машины*. Это не решает проблему одним махом, но разделяет её на две проблемы поменьше. Сначала нам необходимо описать конструкции языка с помощью ограниченного числа операций вычислительной машины определённой архитектуры. После этого остаётся только понять, как работает данная машина. Сама виртуальная машина является формальной абстракцией, так что может быть реализована на любой реальной вычислительной машине.

Многие языки определяются именно таким образом: PL/I (на VDM), PSL [GBM82], Le_Lisp (на LLM3 [Cha80]), Gambit поверх PVM [FM90].

Главная проблема — это разработать хорошую виртуальную машину. Это не так легко, как кажется: она должна одновременно подходить для языка, быть простой в использовании и тривиальной в реализации. На выбор есть множество вариантов: стековые машины, регистровые, основанные на деревьях, графах — в общем, всё, что душе угодно. (Здесь мы сначала выбираем язык ассемблера, а затем подгоняем под него архитектуру машины.) В этом случае сравнение программ сводится к сравнению соответствующих им машинных кодов или же хода их исполнения машиной. Так как в итоге всё упирается в работу вычислительной машины, виртуальной или реальной, то подобный способ определения языка называется *операционной семантикой*.

Но у данного способа есть врождённый недостаток: ему необходима специальная вычислительная машина; иными словами, формальная теория вычислений. Если в качестве такой теории взять какую-нибудь простую, всем понятную концепцию, то можно избавиться от поддержки зоопарка всевозможных машин. Программа — это, прежде всего, функция, преобразующая входные данные в выходные. В этом случае не требуется сложная машина: математика веками оттачивала такой способ «исполнения программ», называя его «применением функций». Таким образом, идея состоит в том, чтобы преобразовать программу в функцию (из определённого множества функций). Подобная функция называется *денотацией* программы. Теперь остаётся только определить вышеупомянутое множество.

Для наших целей прекрасно подойдёт λ-исчисление. Его аксиомы настолько просты, что их толкование не вызывает никаких разногласий. Таким образом, семантику языка программирования можно понимать как способ пере-

вода программ в соответствующие денотации. Этот процесс, между прочим, тоже можно представить как функцию. Денотация программы — это выражение λ-исчисления (λ-терм), представляющее смысл программы. Вооружившись теорией λ-исчисления, мы теперь в состоянии сказать, эквивалентны ли две программы, предсказать результат вычислений и т. д. Конечно, есть множество тонкостей: используемый вариант λ-исчисления, способ определения семантической функции, преобразующей программы в денотации, и многое другое, но общая идея — это именно то, что называется *денотационной семантикой*.

Стоит упомянуть ещё один способ понимания программ, особо полезный для доказательства их корректности. Речь идёт об *аксиоматической семантике* Ричарда Флойда и Тони Хоара. Идея состоит в том, что для всех конструкций языка составляются логические утверждения вида $\{P\}$*форма*$\{Q\}$. Данная формула означает, что если утверждение P истинно перед исполнением *формы*, то в результате её исполнения станет истинным Q. В итоге конструкции языка сводятся к набору подобных утверждений-аксиом, из которых выводятся утверждения-теоремы о поведении программ. Такой подход несомненно удобен для доказательства корректности программ, но, к сожалению, ничего не говорит о том, как реализовывать вычислитель для языка. Ни даже о том, возможно ли построить такой вычислитель в принципе.

Конечно же, многообразие семантик не исчерпывается перечисленными вариантами. Например, существует *естественная семантика* [Kah87], подобная денотационной, но с большим упором на логический вывод свойств программы из аксиом. Или *алгебраическая семантика* [FF89], рассматривающая правила эквивалентных преобразований программ.

5.1. Краткий обзор λ-исчисления

Суть денотационной семантики лежит в определении (специфичной для языка) функции, называемой *интерпретатором* (valuation function), которая переводит корректную программу на языке в соответствующую денотацию — элемент множества денотаций. В качестве подобного множества мы решили взять λ-исчисление, так как оно имеет простую структуру и довольно близко к Scheme, который иногда так и называют «интерпретатором λ-термов» [SS75, Wan84].

Сейчас мы быстро пробежимся по основам λ-исчисления.[1] Его синтаксис очень простой: выражения (термы) ограничиваются переменными, абстракциями и аппликациями. Множество всех доступных переменных будем обозначать **Переменные**. Множество всех термов λ-исчисления будем обозначать Λ; его можно индуктивно определить следующим образом:

[1] Хорошим введением в λ-исчисление могут служить книги [Sto77, Gor88]. Первой книгой библии λ-исчисления считается [Bar84].

переменные:	$\forall x \in$ **Переменные**:	$x \in \Lambda$
абстракции:	$\forall x \in$ **Переменные**, $M \in \Lambda$:	$\lambda x.M \in \Lambda$
аппликации:	$\forall M, N \in \Lambda$:	$(M\ N) \in \Lambda$

Синтаксис не особо важен и термы λ-исчисления одинаково удобно записываются в виде S-выражений.[2] Таким образом, множество термов λ-исчисления синтаксически совпадает с подмножеством программ на Scheme, использующих только одну специальную форму — `lambda`:

x `(lambda (`x`)` M`)` `(`M N`)`

С помощью λ-исчисления мы можем определять функции. Есть даже правило их применения — *β-редукция*: в результате применения функции $\lambda x.M$ к терму N получается новый терм, являющийся телом функции M, в котором вместо переменной x используется N, что коротко записывается: $M[x \to N]$. Это обычная модель подстановки, испокон веков используемая математикой как само собой разумеющееся. Именно так проводятся вычисления в программах на Scheme, принадлежащих вышеупомянутому подмножеству (без побочных эффектов, с одной специальной формой).

$$\beta\text{-редукция:}\quad (\lambda x.M\ N) \xrightarrow{\beta} M[x \to N]$$

Подстановка $M[x \to N]$ должна быть достаточно сообразительной, чтобы нечаянно не затронуть лишние переменные. Имеются в виду свободные переменные, которые могут появиться при подстановке, если M является абстракцией. Свободные переменные терма N не получают значений одноимённых переменных терма M при подстановке N в M; подстановка значений вместо переменных может быть только явной. Говоря более понятным языком, в λ-исчислении принято лексическое связывание. Подобная подстановка определяется следующим образом (скобки отделяют выражения, где происходят изменения):

$x[x \to N] = N$
$y[x \to N] = y \quad$ если $x \neq y$
$(\lambda x.M)[x \to N] = \lambda x.M$
$(\lambda y.M)[x \to N] = \lambda z.\big(M[y \to z][x \to N]\big) \quad$ где $x \neq y$ и z не свободна в M и N
$(M_1\ M_2)[x \to N] = (M_1[x \to N]\ M_2[x \to N])$

Редексом (от reducible expression) или приводимым выражением называется аппликация, где первый терм (терм на месте функции) является абстракцией. β-редукция позволяет избавиться от редексов. Если терм не содержит редексов (является неприводимым), то говорят, что он находится в *нормальной форме*. Термы λ-исчисления не обязательно имеют нормальную форму,

[2] Фактически, именно это Джон Маккарти и сделал в 1960 году.

но если она существует, то, в соответствии с теоремой Чёрча—Россера, она единственна.

Если терм имеет нормальную форму, то она обязательно достижима за конечное число β-редукций. *Стратегией вычислений* называется правило выбора редекса (если их несколько), который будет редуцирован следующим. К сожалению, есть как хорошие правила, так и плохие. Пример хорошего правила: редуцировать самый левый редекс. Оно не обязательно ведёт по кратчайшему пути, но гарантированно достигает нормальной формы (если она вообще существует). Данное правило называется *нормальным* порядком вычислений, он же *вызов по имени*. Примером плохого правила — именно его использует Scheme — является *аппликативный* порядок вычислений, также известный как *вызов по значению* или *энергичный* порядок (eager evaluation). В этом случае функция применяется лишь после вычисления её аргументов. Рассмотрим несколько примеров. Вот терм, который не имеет нормальной формы:

$$(\omega\ \omega) \quad \text{где } \omega = \lambda x.(x\ x) \quad \text{так как } (\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} \dots$$

В Scheme подобная программа приводит к бесконечному циклу и, очевидно, точно не к нормальной форме.

А вот пример терма, который имеет нормальную форму, но правило вычислений, принятое в Scheme, не позволяет её достичь.

$$\big((\lambda x.\lambda y.y\ (\omega\ \omega))\ z\big) \xrightarrow{\beta} (\lambda y.y\ z) \xrightarrow{\beta} z$$

Scheme сразу же пойдёт вычислять аргумент $(\omega\ \omega)$ и попадёт в бесконечный цикл, так и не дойдя до нормальной формы.

Из-за этого правила Scheme также может успешно вычислять термы без нормальной формы: правило требует избавляться от редексов в аргументах сразу же, но запрещает избавляться от них в теле функций. Так что можно спокойно передавать что-то вроде $\lambda x.(\omega\ \omega)$ как аргумент, не получая никаких бесконечных циклов.

Так почему же Scheme использует такое плохое правило? Во-первых, компьютеры гораздо эффективнее обрабатывают вызовы по значению, чем «хорошие» вызовы по имени, даже если их улучшить до *вызовов по необходимости* (также известных как *ленивый* порядок вычислений). Во-вторых, если плохое правило приводит к нормальной форме, то это будет та же форма, которую мы бы получили при использовании хорошего правила. Поэтому ради эффективности Scheme использует плохой аппликативный порядок.

В λ-исчислении принят удобный синтаксический сахар для записи функций нескольких переменных: считать $\lambda xy.M$ сокращением для $\lambda x.\lambda y.M$, а $(M\ N_1\ N_2)$ — сокращением для $((M\ N_1)\ N_2)$. Предыдущий пример становится гораздо понятнее, если его переписать в виде $(\lambda xy.y\ (\omega\ \omega)\ z)$. Теперь бессмысленность вычисления значения локальной переменной x должна быть очевидна: ведь это значение никак не используется в результате.

Естественно, мы легко могли бы говорить про λ-исчисление ещё пару глав, но за этим отправляйтесь к соответствующим книгам: [Bar84, Gor88, Dil88]. Среди всего прочего можно ввести вспомогательные термы, например, целые числа. В результате такого расширения получается *прикладное* λ-исчисление. Мы могли бы также добавить функции от этих термов: например, $2 + 2 \to 4$; они называются δ-правилами. Однако, подобные расширения не являются в строгом смысле необходимыми, так как числа, булевы значения, сложение, вычитание, логическое ИЛИ — всё это можно представить и с помощью обычных λ-термов. Даже списочные структуры вместе с `car`, `cons` и `cdr` возможно представить подобным образом [Gor88]. [**см. упр. 4.2**]

В заключение стоит сказать, что λ-исчисление — это хорошо проработанная теория вычислений, одновременно простая и мощная. β-редукция равна по выразительной силе машине Тьюринга, но при этом не такая запутанная. Также важно, что λ-исчисление обладает при той же простоте более гибким понятием эквивалентных программ: два терма эквивалентны, если эквивалентны их нормальные формы. По всем перечисленным причинам именно λ-исчисление было выбрано множеством денотаций для наших целей.

5.2. Семантика Scheme

По результатам демократических обсуждений, представителем денотаций было избрано λ-исчисление. Предыдущий интерпретатор написан на Scheme без побочных эффектов. Его сердце — это функция-вычислитель `evaluate`, имеющая вот такой тип:

`evaluate`: **Программа** $\times$ **Окружение** $\times$ **Продолжение** $\times$ **Память** $\to$ **Значение**

Первый аргумент (программу) можно легко каррировать, переходя к следующему типу:

Программа $\to$ (**Окружение** $\times$ **Продолжение** $\times$ **Память** $\to$ **Значение**)

Каждая программа превращается в функцию, которая по окружению, продолжению и памяти (вместе они называются *контекстом вычислений*) может сказать нам возвращаемое значение. Это похоже на понятие смысла программы, так что остановимся на таком определении:

интерпретация: **Программа** $\to$ **Денотация**
Денотация: **Окружение** $\times$ **Продолжение** $\times$ **Память** $\to$ **Значение**

Люди, серьёзно занимающиеся денотационной семантикой, не особо любят скобочки и имеют своеобразные предпочтения в наименовании сущностей. Прежде всего, они (повсюду) используют греческие буквы и сокращают всё, что только можно. В итоге, для непосвящённых их записи выглядят загадочным набором иероглифов. Причиной таких привычек служит то, что подобная

запись позволяет уместить семантику языка на одной-единственной странице, где она как на ладони. Это преимущество[3] недостижимо, если использовать пространные названия и лишние символы. Греческие же буквы выбраны для того, чтобы не путать денотации с выражениями определяемого языка. Так как чаще всего денотационная семантика используется для определения языков программирования, алфавит которых ограничен ASCII, то греческие буквы являются хорошим выбором: они компактные и выделяющиеся. Наконец, для уменьшения количества ошибок денотации являются типизированными, их тип выводится из имён переменных.

Мы тоже будем следовать этим соглашениям. По давней традиции функции обозначаются буквой φ. Остальные сущности обычно именуют по первой букве их английского названия: κ для продолжений, α для адресов, ν для идентификаторов (name), π для программ, σ для памяти (store или state). Попробуйте самостоятельно догадаться, почему окружения обозначаются буквой ρ.

π	**Программы**	ρ	**Окружения**
ν	**Переменные**	α	**Адреса**
σ	**Память**	ε	**Значения**
κ	**Продолжения**	φ	**Функции**

Каждое слово, набранное жирным шрифтом, соответствует *домену* объектов. Здесь присутствуют все классы объектов, которые мы рассматривали в предыдущих главах.

$$\begin{aligned}
\textbf{Окружение} &= \textbf{Переменная} \to \textbf{Адрес} \\
\textbf{Память} &= \textbf{Адрес} \to \textbf{Значение} \\
\textbf{Значения} &= \textbf{Функции} + \textbf{Числа} + \textbf{Пары} + \cdots \\
\textbf{Продолжение} &= \textbf{Значение} \times \textbf{Память} \to \textbf{Значение} \\
\textbf{Функция} &= \textbf{Значения}^* \times \textbf{Продолжение} \times \textbf{Память} \to \textbf{Значение}
\end{aligned}$$

Как обычно, звёздочка означает список. Например, домен $\textbf{Значения}^*$ — это домен последовательностей **Значений**. Символ $\times$ означает декартово произведение. Символ $+$ означает дизъюнктивную сумму; то есть **Значение** — это или **Функция**, или **Число**, или **Пара**, и т. д. Это важное свойство дизъюнктивной суммы: каждый элемент домена **Значений** принадлежит одному и только одному из доменов, составляющих дизъюнктивную сумму. Так как мы считаем сущности типизированными, то необходимо чётко обозначать переходы между доменами. Инъекция $\textbf{Значение}(\varepsilon)$ переносит терм ε в домен **Значений**, а проекция $\varepsilon|_{\textbf{Числа}}$ отображает значение ε на домен **Чисел** (конечно же, если ε ему действительно принадлежит).

Домены определяются рекурсивно — это важно с математической точки зрения; именно по этой и некоторым другим причинам они называются доменами, а не просто множествами. Не особо углубляясь в детали, скажем, что

[3] Очень полезное преимущество, если учесть, что средний размер научной публикации — это около десяти страниц.

λ-исчисление было разработано Алонзо Чёрчем в 1930-х годах, но ему не хватало строгой математической модели — она была построена Даной Скоттом около 1970 года. λ-исчисление ещё тогда доказало свою полезность, но с математической моделью оно стало идеальным. Со временем были разработаны и другие модели: D_∞, $\mathcal{P}\omega$. [Sco76, Sto77]

Экстенсиональность это свойство функций $\forall x\colon \big(f(x) = g(x)\big) \Rightarrow (f = g)$. Оно связано с η-редукцией — одним из вспомогательных правил λ-исчисления:

$$\eta\text{-редукция:} \quad \lambda x.(M\ x) \xrightarrow{\eta} M \quad \text{где } x \text{ не свободна в } M$$

Удивительно, но есть как экстенсиональные модели вроде D_∞, так и нет; например, $\mathcal{P}\omega$ не экстенсиональна. Интересно, а наш мир экстенсионален?

Дана Скотт показал, что любая формальная система, рекурсивно определяемая через домены с помощью $\to$, $\times$, $+$ и *, является алгоритмически разрешимой: то есть истинность любого корректного утверждения в ней можно доказать или опровергнуть за конечное число шагов.

Ещё один важный принцип денотационной семантики — это *композициональность*: смысл фрагмента программы зависит только от смысла составляющих его частей. Он очень важен для индуктивного метода доказательства и не только: это удобно для реализации, когда программы не зависят от контекста.

Функцию-интерпретатор обычно обозначают $\mathcal{E}$. Чтобы отличать программы от их семантики, фрагменты программ будут заключаться в семантические скобки: $[\![$ и $]\!]$. Теперь мы наконец-то перейдём к разбору форм одна за другой.

5.2.1. Обращения к переменным

Простейшая из денотаций — получение значения переменной:

$$\mathcal{E}[\![\nu]\!] = \lambda\rho\kappa\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

Денотация ссылки на переменную (с именем ν) это λ-терм, который по окружению ρ, продолжению κ и памяти σ определяет сначала адрес расположения переменной с помощью окружения: $(\rho\ \nu)$, потом передаёт адрес памяти для получения значения: $(\sigma\ (\rho\ \nu))$, затем, наконец, передаёт значение продолжению вместе с исходной памятью (так как чтение её не меняет).

Здесь мы воспользовались упомянутым ранее сокращением для функций нескольких аргументов. Конечно, можно было бы писать более строго:

$$\mathcal{E}[\![\nu]\!] = \lambda\rho.\lambda\kappa.\lambda\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

Нам нет смысла выставлять напоказ свою педантичность, поэтому мы не будем использовать подобную запутанную запись. В действительности, мы её ещё сильнее упростим, приняв синтаксис, похожий на `define`:

$$\mathcal{E}[\![\nu]\!]\rho\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

Конечно, обязательно надо учитывать возможность ошибки: переменной может не оказаться в окружении. С этим будет разбираться начальное окружение ρ_0:

$(\rho_0\ \nu) = \mathit{wrong}$ `"No such variable"`

Если переменная не будет обнаружена в окружении, то вызывается *wrong*, которая в свою очередь вернёт особое значение, обычно обозначаемое $\perp$. Это значение является в некотором смысле абсорбентом, то есть $\forall f\colon f(\perp) = \perp$. Следовательно, если возникает ошибка, то всё вычисление возвращает $\perp$, что явно сигнализирует о проблеме. На самом деле, это не $\perp$ само по себе имеет такое свойство, а функции так с ним обращаются (в этом случае они называются *строгими*). Функция f строга тогда и только тогда, когда $f(\perp) = \perp$. Если условиться использовать только строгие функции, то это избавит нас в определённой мере от обработки ошибок и позволит сконцентрироваться на более важных вещах.

5.2.2. Последовательность

Денотация последовательных вычислений как обычно вызывает вспомогательную функцию, которой в предыдущих интерпретаторах соответствует функция `eprogn`. Мы будем обозначать её $\mathcal{E}^+$. С плюсом, потому что в форме `begin` должен присутствовать хотя бы один элемент. Аналогично будем обозначать последовательность непустых форм: π^+. Интерпретатор $\mathcal{E}^+$ превращает π^+ в денотацию, которая вычисляет все элементы слева направо и возвращает значение последнего из них. Его работа сводится к двум случаям: когда π^+ состоит из одной и более чем одной формы. В Scheme смысл `(begin)` не определён, поэтому и здесь такого случая нет в определении.

$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$
$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$
$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = \big(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma\big)$

Если последовательность состоит из одного элемента, то она эквивалентна данному элементу. Можно было бы записать это буквально:

$\mathcal{E}^+[\![\pi]\!] = \mathcal{E}[\![\pi]\!]$

В λ-исчислении это называется η-упрощением. Мы будем избегать подобных фокусов, так как они затрудняют понимание программ, а также скрывают естественную арность функций; как говорят [WL93], часто так пишут только чтобы выглядеть умнее.

Если последовательность состоит из более чем одного элемента, то вычисляется значение первого из них и управление передаётся продолжению, которое должно вычислить все остальные. Тут — в одной строке! — ещё раз можно увидеть, что продолжение игнорирует вычисленное ранее значение, но не состояние памяти после его вычисления. В конечном итоге именно продолжения позволяют упорядочить вычисления.

5.2.3. Ветвление

Денотация условного оператора довольно стандартна и не представляет трудностей, если знать, как представить булевы значения с помощью абстракций λ-исчисления. На самом деле это не так сложно: мы определим их как комбинаторы (то есть функции без свободных переменных).

$$\mathsf{T} = \lambda xy.x \quad \text{и} \quad \mathsf{F} = \lambda xy.y$$

Эти функции принимают два аргумента и выбирают один из них в качестве результата. Это напоминает логическую операцию **If**, определяемую уравнениями

$$\mathbf{If}(\textit{истина}, p, q) = p \quad \text{и} \quad \mathbf{If}(\textit{ложь}, p, q) = q$$

Внимание: это не имеет ничего общего с `if` в Scheme. В этом определении ничего не говорится о порядке вычислений, только о том, что **If** это булева функция; её можно определить с помощью той же таблицы истинности, но более коротко она описывается вот так:

$$\mathbf{If}(c, p, q) = (\neg c \vee p) \wedge (c \vee q)$$

Благодаря выбранному представлению булевых значений, мы можем легко перенести **If** в λ-исчисление, написав следующий комбинатор:

$$\mathsf{IF}\ c\ p\ q = (c\ p\ q)$$

Как и в логике, здесь ничего не сказано о порядке вычислений, только об отношениях между тремя значениями. Если понимать **If** как функцию, то она возвращает второй аргумент, если первый является истиной, и третий в противном случае. Вслед за [FW84], мы будем называть такую функцию `ef`. На Scheme она бы записывалась так:

```
(define (ef v v1 v2)
  (v v1 v2) )
```

Чтобы сделать запись более понятной, подобный выбор будем записывать как в [Sch86]:

$\varepsilon_0 \to \varepsilon_1\ []\ \varepsilon_2$

В R^{5}RS применяется немного другая нотация:

$\varepsilon_1 \to \varepsilon_2, \varepsilon_3$

Держа всё это в уме, определим денотацию условного оператора:

$\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma =$
$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathit{boolify}\ \varepsilon) \to (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma_1)\ []\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$

Функция *boolify* приводит значение к булевому типу, так как в Scheme любое значение, кроме `#f`, считается истиной. Условный оператор начинает работу с вычисления условия с продолжением, которое выбирает нужную ветку в соответствии со значением условия. Внимание: условный оператор в λ-исчислении только выглядит похожим на *if–then–else* из Scheme, они отличаются очень важной деталью: порядком вычислений. В λ-исчислении он абсолютно произвольный, ничто не мешает параллельно вычислять все три выражения и выбирать одну из двух веток только в конце всех вычислений.

Это очень важное замечание, так как мы не можем считать λ-исчисление (и, следовательно, определяемый нами язык) полностью эквивалентным Scheme. В λ-исчислении вообще нет понятия порядка вычислений, просто условная форма может принять одно из двух значений; какое именно — зависит от первого аргумента.

Тем не менее, отсутствие порядка не мешает правильности вычислений, так как переменные обеих веток изолированы, а функции λ-исчисления не имеют побочных эффектов. Например, следующее выражение вполне успешно вычисляется, хотя кажется, что если вычислять его параллельно или в «неправильном» порядке, то нас ожидает провал:

```
(if (= 0 q) 1 (/ p q))
```

Это выражение сначала проверяет, равна ли `q` нулю, и, если это так, то возвращает единицу, иначе — выполняет деление `p` на `q` и возвращает его результат. Денотация данного вычисления выражает выбор между 1 и p/q в зависимости от того, равна ли q нулю.

Из-за «плохого» порядка вычислений, принятого в Scheme, денотацию этой формы иногда сложно понимать, так как от неё ожидается вычисление условия перед телом, а не вычисление всего сразу вместе с возможными вариантами подобно квантовым компьютерам. Поэтому мы перепишем денотацию следующим образом:

$$\mathcal{E}[\![\texttt{(if } \pi \ \pi_1 \ \pi_2\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.((\mathit{boolify}\ \varepsilon) \to \mathcal{E}[\![\pi_1]\!]\ []\ \mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

Здесь вводится некоторый порядок вычислений. Теперь значение условия служит только для выбора нужной ветки — собственно вычисление значения выполняется отдельно и один раз.

Попробуем проверить, эквивалентны ли данные определения. Для этого достаточно будет доказать эквивалентность их денотаций, иными словами, показать, что

$$(\mathit{boolify}\ \varepsilon) \to (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma)\ []\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma) \equiv ((\mathit{boolify}\ \varepsilon) \to \mathcal{E}[\![\pi_1]\!]\ []\ \mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma)$$

Это тождество очевидно ложно в Scheme из-за аппликативного порядка вычислений. Для доказательства достаточно положить π_2 равной бесконечному циклу. Но денотации — это выражения λ-исчисления, и сравнивать их следует по законам λ-исчисления. В этом случае мы видим простейший дистрибутивный закон.

Надеюсь, иероглифов было достаточно. Далее будет использоваться более читабельная запись условных выражений:

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\textbf{if } (\mathit{boolify}\ \varepsilon) \textbf{ then } \mathcal{E}[\![\pi_1]\!] \textbf{ else } \mathcal{E}[\![\pi_2]\!] \textbf{ endif}\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

5.2.4. Присваивание

Денотация присваивания проста. Версия, приведённая здесь, возвращает только что присвоенное значение.

$$\mathcal{E}[\![\texttt{(set! } \nu\ \pi\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \to \varepsilon])\ \sigma)$$

$$f[y \to z] = \lambda x.\textbf{if } y = x \textbf{ then } z \textbf{ else } (f\ x) \textbf{ endif}$$

Память расширяется, чтобы учесть новое значение ε по адресу переменной ν. Мы обозначаем это расширение $\sigma[\alpha \to \varepsilon]$. Подобная нотация используется и в других работах: $[\alpha \to \varepsilon]\sigma$ в [Sch86], $[\varepsilon/\alpha]\sigma$ в [Sto77] или $\sigma[\varepsilon/\alpha]$ в [Gor88, KCR98].

Давайте расширим λ-исчисление несколькими вспомогательными функциями. Последовательности будем записывать в угловых скобках: $\langle$ и $\rangle$. Конкатенацию последовательностей будем обозначать знаком §. Извлечение i-го элемента последовательности будем обозначать $\langle\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\rangle\downarrow_i$ (получая ε_i). Отбрасывание первых i элементов последовательности — $\langle\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\rangle\uparrow_i$ (получая $\langle\varepsilon_{i+1}, \ldots, \varepsilon_n\rangle$). Длина последовательности ε^* записывается $\#\varepsilon^*$. Все эти определения можно было бы привести непосредственно в виде λ-термов, но это стало бы чересчур серьёзным отступлением от темы, поэтому, ничуть не умаляя достоверности повествования, мы сходу будем использовать $\downarrow_1$, $\uparrow_1$, $\#$ и § как денотационные эквиваленты `car`, `cdr`, `length` и `append`.

Теперь можно коротко записать расширение памяти списком переменных. Предполагается, что y^* и z^* имеют равную длину.

$$f[y^* \xrightarrow{*} z^*] = \textbf{if } \#y^* > 0 \textbf{ then } f[y^*\uparrow_1 \xrightarrow{*} z^*\uparrow_1][y^*\downarrow_1 \to z^*\downarrow_1] \textbf{ else } f \textbf{ endif}$$

5.2.5. Абстракция

Для начала рассмотрим лишь случай функций фиксированной арности.

$$\begin{aligned}&\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{) }\pi^+\texttt{)}]\!]\rho\kappa\sigma = \\ &\quad(\kappa\ \textbf{Значение}(\lambda\varepsilon^*\kappa_1\sigma_1.\textbf{if } \#\varepsilon^* = \#\nu^* \\ &\qquad\qquad\textbf{then } \mathit{allocate}\ \sigma_1\ \#\nu^* \\ &\qquad\qquad\qquad\lambda\sigma_2\alpha^*.(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\ &\qquad\qquad\textbf{else } \mathit{wrong}\ \texttt{"Incorrect arity"} \\ &\qquad\qquad\textbf{endif})\ \sigma)\end{aligned}$$

Инъекция **Значение**(...) принимает λ-терм, представляющий функцию, и превращает его в значение. При применении функции производится обратное преобразование.

Вызванная функция проверят фактическую арность, затем выделяет место в памяти под свои аргументы и связывает переменные со значениями, после чего последовательно вычисляет формы, составляющие её тело. Выделением памяти занимается функция *allocate*; она принимает память, необходимое количество адресов и продолжение, которому она передаст список выделенных адресов и новое состояние памяти. *allocate* — это чистая функция: эквивалентные аргументы дают эквивалентные результаты её применения. Её определение довольно рутинно, так что здесь мы его опустим, чтобы не перегружать выкладки. (Интересующиеся могут переписать определение *allocate* из предыдущей главы. [**см. стр. 167**])

Функция *allocate* имеет следующий полиморфный тип (α соответствует любому типу данных):

$$\textbf{Память} \times \textbf{НатуральноеЧисло} \times (\textbf{Память} \times \textbf{Адреса}^* \to \alpha) \to \alpha$$

5.2.6. Аппликация

Функции создаются для того, чтобы их применяли, поэтому на очереди аппликация. Как обычно, нам потребуется вспомогательный интерпретатор: $\mathcal{E}^*$, аналог `evlis`.

$$\mathcal{E}[\![(\pi\ \ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi|_{\textbf{Функции}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$
$$\mathcal{E}^*[\![\,]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$
$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\ \S\ \varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

Продолжения, которые принимает $\mathcal{E}^*$, работают со списками значений, а не с отдельными значениями. Это отличает $\mathcal{E}^*$ от $\mathcal{E}^+$. Данные продолжения κ имеют тип

$$\textbf{Значения}^* \times \textbf{Память} \to \textbf{Значение}$$

5.2.7. `call/cc`

Разумеется, наше знакомство с денотационной семантикой было бы неполным без определения существенной для Scheme функции — `call/cc`. Глобальная функция `call/cc` определяется следующим образом:

$$\begin{aligned}&(\sigma_0\ (\rho_0\ [\![\texttt{call/cc}]\!])) = \\ &\quad\textbf{Значение}(\lambda\varepsilon^*\kappa\sigma.\textbf{if}\ \#\varepsilon^* = 1 \\ &\qquad\qquad\textbf{then}\ (\varepsilon^*{\downarrow_1}|_{\textbf{Функции}} \\ &\qquad\qquad\quad\langle\textbf{Значение}(\lambda\varepsilon_1^*\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon_1^* = 1\ \textbf{then}\ (\kappa\ \varepsilon_1^*{\downarrow_1}\ \sigma_1) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\textbf{else}\ \textit{wrong}\ \texttt{"Incorrect arity"} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\textbf{endif})\rangle\ \kappa\ \sigma)\end{aligned}$$

else *wrong* `"Incorrect arity"`
endif)

Обратите внимание на переходы между доменами **Значений** и **Функций**. Сама денотация не сложнее любого другого из способов определения `call/cc`, которые были рассмотрены ранее в третьей главе. [**см. стр. 124**]

5.2.8. Предварительные выводы

Нам удалось шаг за шагом определить функцию, которая ставит в соответствие каждой программе λ-терм. Не подлежит сомнению тот факт, что она существует, ведь мы руководствовались принципом композициональности при её определении. Возможны только некоторые проблемы с синтаксически рекурсивными программами [Que92a], так что для точного доказательства корректности потребуется немного больше математики.

Теперь вы можете своими глазами убедиться в том, что семантика ядра Scheme умещается на одном листе бумаги: см. таблицу 5.1.

Конечно, специальная форма `quote` остаётся нереализованной (как вы помните, с цитированием не всё так просто [**см. стр. 175**]), функции с переменной арностью отложены на потом. Естественно, здесь не хватает `eq?` для сравнения функций, а также множества других примитивов. Правда, у нас есть `call/cc` в костюме Евы. На самом деле важно то, что другие функции вроде `car`, `cons`, `set-cdr!` могут быть реализованы с помощью данного базиса без каких-либо добавок в ядро. Специальных форм и некоторых примитивных функций вроде `call/cc` достаточно для понимания принципа.

Как бы то ни было, я настаиваю на том, что выражение сути Scheme с подобной точностью и лаконичностью стоит затраченных усилий. Таким образом, наше путешествие, начавшееся в первой главе с определения подмножества Scheme с помощью всего Scheme, подходит к концу. Сейчас мы определили весь Scheme с помощью одного лишь λ-исчисления.

5.3. Семантика λ-исчисления

Определение сущности языка с помощью нескольких формул является одной из притягательных черт Scheme. Именно поэтому функциональные языки являются лингвистическими лабораториями для испытания новых конструкций и изучения их фундаментальных, обобщённых свойств. Попробуем и мы провести подобное исследование. Для этого мы примемся за довольно простой по своей сути язык — само λ-исчисление. Здесь нет никакой тавтологии или шутки, будет полезным рассмотреть семантику языка, который отличается от Scheme, но всё же является родственным.

Начнём с синтаксиса. Он, как уже было сказано, не важен, так что будем использовать Scheme-подобный синтаксис:

x `(lambda (`x`)` M`)` `(`M N`)`

$$\mathcal{E}[\![\nu]\!]\rho\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\textbf{if}\ (\mathit{boolify}\ \varepsilon)\ \textbf{then}\ \mathcal{E}[\![\pi_1]\!]\ \textbf{else}\ \mathcal{E}[\![\pi_2]\!]\ \textbf{endif}\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \to \varepsilon])\ \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma =$$
$$(\kappa\ \textbf{Значение}(\lambda\varepsilon^*\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* = \#\nu^*\ \textbf{then}\ \mathit{allocate}\ \sigma_1\ \#\nu^*\ \lambda\sigma_2\alpha^*.(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi|_{\textbf{Функции}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\ \S\ \varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\,]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = \big(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma\big)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$$

$$(\sigma_0\ (\rho_0\ [\![\texttt{call/cc}]\!])) =$$
$$\textbf{Значение}(\lambda\varepsilon^*\kappa\sigma.\ \textbf{if}\ \#\varepsilon^* = 1\ \textbf{then}\ (\varepsilon^*{\downarrow_1}|_{\textbf{Функции}}\ \langle\textbf{Значение}(\lambda\varepsilon_1^*\kappa_1\sigma_1.\ \textbf{if}\ \#\varepsilon_1^* = 1\ \textbf{then}\ (\kappa\ \varepsilon_1^*{\downarrow_1}\ \sigma_1)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\rangle\ \kappa\ \sigma)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})$$

Таблица 5.1. Сущность Scheme.

Теперь определим домены. В λ-исчислении нет присваиваний и продолжений, что серьёзно облегчает нам работу. Более того, мы ограничимся чистым λ-исчислением — то есть никаких чисел и всего такого, только голые абстракции. Итак, домены:

$$\begin{array}{ll} \pi & \textbf{Программы} \\ \nu & \textbf{Переменные} \\ \rho & \textbf{Окружения} = \textbf{Переменная} \to \textbf{Значение} \\ \varepsilon & \textbf{Значения} \quad = \textbf{Функции} \\ \varphi & \textbf{Функции} \quad = \textbf{Значение} \to \textbf{Значение} \end{array}$$

Функцию-интерпретатор мы назовём $\mathcal{L}$. Она будет сопоставлять каждому λ-терму его денотацию — другой λ-терм. Таким образом, интерпретатор имеет простой тип

$$\mathcal{L}\colon \quad \textbf{Программа} \to (\textbf{Окружение} \to \textbf{Значение})$$

Осталось только определить одну за другой денотации синтаксических форм. Всё их огромное разнообразие собрано в таблице 5.2.

$$\mathcal{L}[\![\nu]\!]\rho = (\rho\ \nu)$$
$$\mathcal{L}[\![\texttt{(lambda (}\nu\texttt{) }\pi\texttt{)}]\!]\rho = \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \to \varepsilon])$$
$$\mathcal{L}[\![\texttt{(}\pi\ \pi'\texttt{)}]\!]\rho = \big((\mathcal{L}[\![\pi]\!]\ \rho)\ (\mathcal{L}[\![\pi']\!]\ \rho)\big)$$

Таблица 5.2. Семантика λ-исчисления.

Данная семантика очень точна, вплоть до неопределённости порядка вычислений. Аппликация переводится в аппликацию, так что никакого порядка не устанавливается.

Интерпретатор $\mathcal{L}$ определяется рекурсивно. Мы можем так делать благодаря композициональности: рано или поздно все вычисления сведутся к ссылкам на переменные.

λ-исчисление представляет особенный случай, так как мы и безо всяких денотаций имеем довольно чёткое представление о его семантике. Можно доказать [Sto77], что подобное самоопределение сохраняет все необходимые свойства λ-исчисления вроде β-редукции.

Денотирование λ-исчислением хорошо работает для языков без побочных эффектов и продолжений. Но если данные явления в языке всё же присутствуют, то эффективнее будет использовать другой метод, в котором можно явно задавать порядок вычислений. Также для работы присваивания необходимо разделить окружение и память, введя механизм ссылок, как в ML, коробки из четвёртой главы, или нечто подобное. [**см. стр. 145**]

5.4. Функции с переменной арностью

В этом разделе мы покажем, как привнести в Scheme функции с точечным аргументом. Особенность этих функций в том, что они собирают «лишние» аргументы в список, который передаётся через последний аргумент. Следовательно, при каждом вызове подобной функции необходимо создавать в памяти список, что влияет на производительность. От некоторых проблем нас может избавить функция `apply`, которая сразу принимает список аргументов и применяет к нему функцию, но ведь эти списки тоже кто-то должен создать. В общем, переменная арность неразлучно связана со списками, так что нам потребуются соответствующие функции: `car`, `cons` и т. д.

Точечные пары принадлежат домену **Пар**. Естественно, этот домен также входит в дизъюнктивную сумму **Значений**. Точечные пары представляются так же, как и в предыдущей главе: двумя адресами.

$$\textbf{Значения} = \textbf{Функции} + \textbf{Числа} + \textbf{Пары} + \cdots$$
$$\textbf{Пара} = \textbf{Адрес} \times \textbf{Адрес}$$

Денотации `cons`, `car` и `set-cdr!` (надо же показать хотя бы один побочный эффект) не таят сюрпризов; мы используем в точности тот же подход, что и в предыдущей главе. Единственное, о чём стоит договориться, это ассоциативность операций. Выражение $\varepsilon^*{\downarrow_1}|_{\textbf{Пары}}{\downarrow_2}$ читается слева направо: первый аргумент $\varepsilon^*{\downarrow_1}$, пока ещё значение, проецируется на домен пар $|_{\textbf{Пары}}$ (если это значение на самом деле не было парой, то мы получаем $\bot$), после чего извлекается `cdr` этой пары, её второй компонент $\downarrow_2$.

$(\sigma_0\ (\rho_0\ [\![\texttt{cons}]\!])) =$
  Значение$(\lambda\varepsilon^*\kappa\sigma.$**if** $\#\varepsilon^* = 2$
    then *allocate* σ 2
      $\lambda\sigma_1\alpha^*.(\kappa$ **Значение**$(\langle\alpha^*{\downarrow_1}, a^*{\downarrow_2}\rangle)\ \sigma_1[\alpha^* \xrightarrow{*} \varepsilon^*])$
    else *wrong* `"Incorrect arity"`
    endif$)$

$(\sigma_0\ (\rho_0\ [\![\texttt{car}]\!])) =$
  Значение$(\lambda\varepsilon^*\kappa\sigma.$**if** $\#\varepsilon^* = 1$
    then $(\kappa\ (\sigma\ \varepsilon^*{\downarrow_1}|_{\textbf{Пары}}{\downarrow_1})\ \sigma)$
    else *wrong* `"Incorrect arity"`
    endif$)$

$(\sigma_0\ (\rho_0\ [\![\texttt{set-cdr!}]\!])) =$
  Значение$(\lambda\varepsilon^*\kappa\sigma.$**if** $\#\varepsilon^* = 2$
    then $(\kappa\ \varepsilon^*{\downarrow_1}\ \sigma[\varepsilon^*{\downarrow_1}|_{\textbf{Пары}}{\downarrow_2} \to \varepsilon^*{\downarrow_2}])$
    else *wrong* `"Incorrect arity"`
    endif$)$

После определения структуры списков написание `apply` не составляет проблем. Мы собираем в последовательность все элементы формы аппликации,

кроме первого (функции) и последнего. Если предпоследний элемент является списком, то его необходимо пришить к концу формируемой последовательности. После всех этих манипуляций выполняется собственно вызов функции.

$$
\begin{array}{l}
(\sigma_0\ (\rho_0\ [\![\texttt{apply}]\!])) = \\
\quad \textbf{Значение}(\lambda\varepsilon^*\kappa\sigma.\textbf{if}\ \#\varepsilon^* \geqslant 2 \\
\qquad\qquad \textbf{then}\ (\varepsilon^*{\downarrow_1}|_{\textbf{Функции}}\ (\mathit{collect}\ \varepsilon^*{\uparrow_1})\ \kappa\ \sigma) \\
\qquad\qquad\quad \textbf{where}\ \mathit{collect} = \lambda\varepsilon_1^*.\textbf{if}\ \varepsilon_1^*{\uparrow_1} = \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\mathit{flat}\ \varepsilon_1^*{\downarrow_1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \langle\varepsilon_1^*{\downarrow_1}\rangle\ \S\ (\mathit{collect}\ \varepsilon_1^*{\uparrow_1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\quad \textbf{and}\ \mathit{flat} = \lambda\varepsilon.\textbf{if}\ \varepsilon \in \textbf{Пары} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \langle(\sigma\ \varepsilon|_{\textbf{Пары}}{\downarrow_1})\rangle\ \S\ (\mathit{flat}\ (\sigma\ \varepsilon|_{\textbf{Пары}}{\downarrow_2})) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \langle\rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif} \\
\qquad\qquad \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"} \\
\qquad\qquad \textbf{endif})
\end{array}
$$

Здесь **where** ... **and** ... определяет взаимно рекурсивные функции.

Теперь перейдём к собственно обработке точечных аргументов. Для этого нам потребуется изменить форму `lambda`, а также ввести ещё один интерпретатор. Его задачей будет правильным образом связывать переменные и значения. Мы назовём его $\mathcal{B}$, от *binding*. Его тип связан с типом $\mathcal{E}$ для функций; для краткости обозначим буквой μ денотацию функции, для которой $\mathcal{B}$ подготавливает аргументы, а для контекста вычислений используем букву τ:

$$
\begin{array}{rl}
 & \tau \equiv \textbf{Значения}^* \times \textbf{Окружение} \times \textbf{Продолжение} \times \textbf{Память} \\
\mathcal{E}: & \textbf{Программа} \to (\tau \to \textbf{Значение}) \\
\mathcal{B}: & \textbf{СписокАргументов} \to \big(\underbrace{(\tau \to \textbf{Значение})}_{\mu} \times \tau \to \textbf{Значение}\big)
\end{array}
$$

$\mathcal{B}$, связывающий интерпретатор, начинает работу после проверки арности. Если с ней всё в порядке, то он заносит один за другим фактические значения аргументов в память и расширяет лексическое окружение функции соответствующими переменными. Наконец, управление передаётся телу функции. Определение функций фиксированной арности с помощью $\mathcal{B}$ выглядит так:

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\
\quad (\kappa\ \textbf{Значение}(\lambda\varepsilon^*\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* = \#\nu^* \\
\qquad\qquad\qquad \textbf{then}\ \big((\mathcal{B}[\![\nu^*]\!]\ \lambda\varepsilon_1^*\rho_1\kappa_2\sigma_2.(\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2))\ \varepsilon^*\ \rho\ \kappa_1\ \sigma_1\big) \\
\qquad\qquad\qquad \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"} \\
\qquad\qquad\qquad \textbf{endif})\ \sigma) \\
\mathcal{B}[\![\nu\ \nu^*]\!]\mu = (\mathcal{B}[\![\nu]\!]\ (\mathcal{B}[\![\nu^*]\!]\ \mu)) \\
\mathcal{B}[\![\,]\!]\mu = \mu \\
\mathcal{B}[\![\nu]\!]\mu = \lambda\varepsilon^*\rho\kappa\sigma.\mathit{allocate}\ \sigma\ 1\ \lambda\sigma_1\alpha^*.\textbf{let}\ \alpha = \alpha^*{\downarrow_1} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mu\ \varepsilon^*{\uparrow_1}\ \rho[\nu \to \alpha]\ \kappa\ \sigma_1[\alpha \to \varepsilon^*{\downarrow_1}])
\end{array}
$$

Конструкция **let** ... **in** вводит локальные нерекурсивные определения.

Для обработки случая функций переменной арности потребуется определить дополнительный синтаксис формы `lambda`, принимающей список с точкой, а также новый вариант $\mathcal{B}$. Эта форма принимает последовательность «лишних» значений, превращает её в список (настоящий, из точечных пар), который связывает с именем последнего аргумента. Если учесть, что функции с переменной арностью и `apply` могут использоваться совместно с побочными эффектами, то подобные списки необходимо создавать заново при каждом вызове. В конце концов, в стандарте есть требование, чтобы функции вели таким образом. Поэтому следующее выражение должно возвращать ложь:

```
(let ((arguments (list 1 2 3)))
  (apply (lambda args (eq? args arguments)) arguments) )
```

Если мы не хотим этого делать, то вначале потребуется доказать, что совместное использование одного и того же физического списка не изменит смысла программы.

Наконец, переходим к денотациям:

$$
\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}\nu^*\ \texttt{.}\ \nu\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\
&\quad(\kappa\ \textbf{Значение}(\lambda\varepsilon^*\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* \geqslant \#\nu^* \\
&\qquad\qquad\textbf{then}\ \big((\mathcal{B}[\![\nu^*]\!]\ (\mathcal{B}[\![\texttt{.}\ \nu]\!]\ \lambda\varepsilon_1^*\rho_1\kappa_2\sigma_2. \\
&\qquad\qquad\qquad\qquad(\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2)))) \\
&\qquad\qquad\quad\varepsilon^*\ \rho\ \kappa_1\ \sigma_1\big) \\
&\qquad\qquad\textbf{else}\ \textit{wrong}\ \texttt{"Incorrect arity"} \\
&\qquad\qquad\textbf{endif})\ \sigma) \\
&\mathcal{B}[\![\texttt{.}\ \nu]\!]\mu = \lambda\varepsilon^*\rho\kappa\sigma.\big(\textit{listify}\ \varepsilon^*\ \sigma\ \lambda\varepsilon\sigma_1. \\
&\qquad\qquad\qquad\qquad\textit{allocate}\ \sigma_1\ 1 \\
&\qquad\qquad\qquad\qquad\lambda\sigma_2\alpha^*.\textbf{let}\ \alpha = \alpha^*{\downarrow_1} \\
&\qquad\qquad\qquad\qquad\qquad\textbf{in}\ (\mu\ \langle\rangle\ \rho[\nu \to \alpha]\ \kappa\ \sigma_2[\alpha \to \varepsilon])\big) \\
&\textbf{where}\ \textit{listify} = \lambda\varepsilon_1^*\sigma_1\kappa_1. \\
&\qquad\qquad\textbf{if}\ \#\varepsilon_1^* > 0 \\
&\qquad\qquad\textbf{then}\ \textit{allocate}\ \sigma_1\ 2 \\
&\qquad\qquad\qquad\lambda\sigma_2\alpha^*.\textbf{let}\ \kappa_2 = \lambda\varepsilon\sigma_3. \\
&\qquad\qquad\qquad\qquad\qquad(\kappa_1\ \textbf{Значение}(\alpha^*)\ \sigma_3[\alpha^*{\downarrow_2} \to \varepsilon]) \\
&\qquad\qquad\qquad\qquad\textbf{in}\ (\textit{listify}\ \varepsilon_1^*{\uparrow_1}\ \sigma_2[\alpha^*{\downarrow_1} \to \varepsilon_1^*{\downarrow_1}]\ \kappa_2) \\
&\qquad\qquad\textbf{else}\ (\kappa_1\ \textbf{Значение}(\langle\rangle)\ \sigma_1) \\
&\qquad\qquad\textbf{endif}
\end{aligned}
$$

Как видите, введение нетривиальных возможностей вроде функций переменной арности требует существенных усилий. Нам пришлось, фактически, написать ещё один интерпретатор. Если сравнить получившуюся версию с элегантной реализацией ядра Scheme, то станет очевидным, что добавление всего одной интересной, но всё же несущественной детали почти удвоило размер кода, не говоря уже о читабельности и понятности.

5.5. Порядок вычисления аргументов

Время от времени в Интернете вспыхивают яростные религиозные войны по поводу толкования стандартов Scheme. Данная книга строго следует стандарту: порядок вычисления термов аппликации не определён. К сожалению, из-за этого правила сложно не только писать корректные программы, зависящие от конкретного порядка, но и отлаживать некорректные. Многие программы, даже некоторые из написанных видными экспертами, неявно зависят от порядка вычислений; особенно программы, использующие продолжения. Лично я предпочитаю порядок слева направо, так как он соответствует направлению чтения во многих языках, а также привносит хоть какую-то систематичность в поиск ошибок.

Особый интерес среди доводов противоборствующей стороны представляют следующие два. Первый из них состоит в том, что многие языки не указывают порядок вычислений. Даже императивный Си не указывает порядок вычисления аргументов функций. Если выражение `(foo (f x) (g x y))` транслируется в следующий код на Си: `foo(f(x), g(x, y))`, то нам действительно не следует полагаться на какой-либо порядок.

Второй довод больше философский. В языке без предопределённого порядка вычислений его всё же можно явно установить самостоятельно с помощью `begin`. Следовательно, порядок всё же есть, просто «не определён» следует читать как «не определён нами». Поэтому для истинной неопределённости реализация языка должна использовать какой-нибудь генератор случайных чисел, который будет определять «неопределённый» порядок непосредственно перед вычислениями. [**см. упр. 5.1**]

В Си порядок в действительности не определяется для того, чтобы разрешить компилятору выбирать любой необходимый ему порядок, например, чтобы эффективнее распределять регистры.

Явный порядок вычислений упрощает отладку, делая одной неопределённостью меньше. Если порядок будет случайным, то два прогона одной и той же программы на одних и тех же данных могут дать разные результаты. Рассмотрим пример[4]:

```
(define (dynamically-changing-evaluation-order?)
  (define (amb)
    (call/cc (lambda (k) ((k #t) (k #f)))) )
  (if (eq? (amb) (amb))
    (dynamically-changing-evaluation-order?)
    #t ) )
```

Функция `amb` возвращает истину или ложь в зависимости от того, какой терм вычисляется первым. Если порядок вычислений не фиксирован, то функция `dynamically-changing-evaluation-order?` рано или поздно остановится

[4] Данная функция была придумана и реализована совместно с Матиасом Феллайзеном (Matthias Felleisen).

и вернёт `#t`; иначе она попадает в бесконечный цикл. В R^{5}RS нет ничего, что требовало бы от этой функции того или иного поведения. Оно зависит исключительно от реализации.

Необходимо чётко отделять реализацию языка от его стандарта. Интерпретатор или компилятор по очевидным причинам вынуждены использовать хоть какой-нибудь порядок. Не особо важно, слева направо, или справа налево, как MacScheme, или хоть в зависимости от текущей фазы Луны. Но ни одна из известных мне реализаций не изменяет всерьёз порядок вычисления аргументов во время выполнения программы. Обычно порядок устанавливается при компиляции, после чего к этому вопросу больше не возвращаются. Поэтому неопределённость порядка всё же следует понимать как привилегию выбирать любой, а не обязанность поддерживать случайный.

Проблема в том, как показать в денотациях неопределённость порядка, но в то же время разрешить реализациям определять свой порядок, используя одни и те же денотации. Идея решения данной проблемы состоит в небольшом изменении структуры денотаций. Сейчас они считаются λ-термами, которые принимают окружение, продолжение и память, а потом возвращают некоторый результат. Данный результат принадлежит домену **Значений**, но ведь на самом деле он неразрывно связан с состоянием памяти: мы говорили, что побочные эффекты — это тоже результат работы программы. Поэтому нам стоит сделать результатом вычислений именно пару (*значение*, *память*). Мы не будем заниматься философствованиями на тему того, чем именно является результат вычислений, а просто сделаем областью значений денотаций домен **Результатов**.

Так как вычисления могут проходить по-разному, то и денотации будут возвращать не один результат, а множество всех возможных результатов в зависимости от порядка вычислений. Реализация же будет выбирать из этого множества какой-нибудь один, руководствуясь собственными соображениями. Следовательно, $\mathcal{E}$ теперь соответствует обобщённому результату, а конкретная реализация представляется функцией $\mathcal{N}$. Если обозначить множество всех подмножеств Q как $\mathcal{P}(Q)$, то эти функции имеют следующие типы:

$$\mathcal{E}: \quad \textbf{Программа} \to \textbf{Значения}^* \times \textbf{Окружение} \times \\ \times \textbf{Продолжение} \times \textbf{Память} \to \mathcal{P}(\textbf{Результаты})$$

$$\mathcal{N}: \quad \textbf{Программа} \to \textbf{Значения}^* \times \textbf{Окружение} \times \\ \times \textbf{Продолжение} \times \textbf{Память} \to \textbf{Результат}$$

$\mathcal{N}$ определяется просто: она использует *oneof*, чтобы выбрать один из возможных результатов. Определение функции *oneof*, естественно, возлагается на реализацию.

$\mathcal{N}[\![\pi]\!]\rho\kappa\sigma = (oneof\ (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma))$

Теперь необходимо переопределить аппликацию функций так, чтобы получать все возможные результаты. Если порядок случаен, то это фактически значит, что аппликация выполняется так: сначала выбирается случайный

терм, обозначим его π'_0, он вычисляется в ε'_0, потом из оставшихся выбирается следующий терм π'_1, который вычисляется в ε'_1, и так далее. Затем значения $\varepsilon'_0, \varepsilon'_1, \ldots, \varepsilon'_n$ переупорядочиваются так, как они шли в исходной форме: $\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_n$, и, наконец, первое значение-функция применяется к последовательности из всех остальных. При таком подходе порядок вычислений действительно случаен для каждого отдельного вызова, а не выбирается и фиксируется для каждой функции. Рассмотрим пример. Следующая функция не только выводит неопределённое число, но и продолжение, которое она возвращает, тоже выводит неопределённое число.

```
(define (one-two-three)
  (call/cc (lambda (k)
             ((begin (display 1) (call/cc k))
              (begin (display 2) (call/cc k))
              (begin (display 3) (call/cc k)) ) )) )
```

Денотация аппликации с неопределённым порядком вычисления аргументов будет выполнять именно то, что мы сказали ранее. Она рассмотрит все возможные перестановки аргументов с помощью функции *forall*, которая применяет свой первый аргумент (тернарную функцию) ко всем возможным срезам своего второго аргумента (списка). Срезы предоставляются функцией *cut*, которая разрезает список на две части: одна из них содержит первые i элементов, другая — все оставшиеся; затем *cut* применяет свой третий аргумент (продолжение) к этим двум частям. Определения слегка запутанные, но это хороший пример стиля передачи продолжений. (Данная программа была разработана в сотрудничестве с Софи Англад и Жан-Жаком Лакрампом [ALQ95].)

$$\mathcal{E}[\![(\pi_0\ \pi_1\ \ldots\ \pi_n)]\!]\rho\kappa\sigma = \big((\mathit{possible\text{-}paths}\ \langle \mathcal{E}[\![\pi_0]\!], \mathcal{E}[\![\pi_1]\!], \ldots, \mathcal{E}[\![\pi_n]\!]\rangle)$$
$$\rho\ \lambda\varepsilon^*\sigma_1.(\varepsilon^*{\downarrow_1}|_{\textbf{Функции}}\ \varepsilon^*{\uparrow_1}\ \kappa\ \sigma_1)\ \sigma\big)$$

$$\begin{array}{l}
(\mathit{possible\text{-}paths}\ \mu^+) = \\
\quad \lambda\rho\kappa\sigma.\textbf{if}\ \#\mu^+{\uparrow_1} > 0 \\
\qquad \textbf{then}\ (\mathit{forall}\ \lambda\mu_1^+\mu\mu_2^+. \\
\qquad\qquad (\mu\ \rho\ \lambda\varepsilon\sigma_1. \\
\qquad\qquad\quad ((\mathit{possible\text{-}paths}\ \mu_1^+ \S\ \mu_2^+) \\
\qquad\qquad\quad \rho\ \lambda\varepsilon^*\sigma_2.\textbf{let}\ \kappa_1 = \lambda\varepsilon_1^*\varepsilon_2^*. \\
\qquad\qquad\qquad\qquad (\kappa\ \varepsilon_1^* \S \langle\varepsilon\rangle \S \varepsilon_2^*\ \sigma_2) \\
\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{cut}\ \#\mu_1^+\ \varepsilon^*\ \kappa_1) \\
\qquad\qquad\quad \sigma_1)\ \sigma)\ \mu^+) \\
\qquad \textbf{else}\ (\mu^+{\downarrow_1}\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \langle\varepsilon\rangle\ \sigma_1)\ \sigma) \\
\qquad \textbf{endif}
\end{array}$$

$$\begin{array}{l}
(\mathit{forall}\ \varphi\ \ell) = (\mathit{loop}\ \langle\rangle\ \ell{\downarrow_1}\ \ell{\uparrow_1}) \\
\quad \textbf{where}\ \mathit{loop} = \lambda\ell_1\varepsilon\ell_2.(\varphi\ \ell_1\ \varepsilon\ \ell_2) \cup \textbf{if}\ \#\ell_2 > 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\mathit{loop}\ \ell_1 \S \langle\varepsilon\rangle\ \ell_2{\downarrow_1}\ \ell_2{\uparrow_1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \varnothing \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif}
\end{array}$$

$$
\begin{aligned}
&(cut\ \iota\ \varepsilon^*\ \kappa) = (accumulate\ \iota\ \langle\rangle\ \varepsilon^*)\\
&\ \textbf{where}\ accumulate = \lambda\iota\ell\ell_1.\textbf{if}\ \iota > 0\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (accumulate\ (\iota - 1)\ \langle \ell_1{\downarrow_1}\rangle\ \S\ \ell\ \ell_1{\uparrow_1})\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ (reverse\ \ell)\ \ell_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif}
\end{aligned}
$$

Во всех приведённых случаях порядок вычисления аргументов является неопределённым, но сами вычисления остаются последовательными — это требование стандарта: результат должен быть эквивалентен вычислению аргументов в какой-нибудь последовательности. То есть следующая программа может вернуть `(3 5)` или `(4 3)`, но никак не `(3 3)`:

```
(let ((x 1) (y 2))
  (list (begin (set! x (+ x y)) x)
        (begin (set! y (+ x y)) y) ) )
```

Денотация следующей программы, полученная с помощью новой $\mathcal{E}$, будет возвращать два возможных результата: 1 и 2. Конкретная реализация сможет выбрать один из них с помощью $\mathcal{N}$ и *oneof*.

```
(call/cc (lambda (k) ((k 1) (k 2)))) → {1, 2}
```

5.6. Динамическое связывание

Идея динамического связывания не только важна сама по себе, но и действительно полезна. В конце концов, довольно долгое время разнообразные диалекты Лиспа были именно динамическими. Поэтому мы просто не можем обойти стороной эту идею и не показать её денотацию. Для этого нам потребуется поправить ядро Scheme, а также добавить несколько специальных форм для работы с новым типом привязок. Конечно, есть далеко не один вариант реализации динамического окружения; взять хотя бы возможность использовать или специальные формы, или же специализированные функции. [**см. стр. 72**] Традиционно в Scheme принят второй подход, чтобы как можно меньше вмешиваться в ядро языка. К сожалению, не все функции можно отделить от ядра, даже если они вызываются, используются и ведут себя именно как функции. Например, `call/cc`: ей необходим доступ к продолжениям. Аналогично, всем функциям для работы с динамическими переменными потребуется доступ к окружению динамических переменных. Доработанное ядро Scheme приведено в таблице 5.3.

Динамическое окружение, обозначаемое δ, отличается от лексического ρ. Оно передаётся только туда, где его можно использовать. Отличается оно и от памяти σ, которая является *однопоточной*: каждое вычисление принимает память, достаёт значения нужных переменных или изменяет их, после чего передаёт новое состояние памяти дальше. Поток передачи памяти только один, в каждый момент времени существует только одно актуальное состояние памяти. Динамическое окружение хоть и передаётся от функции к функции

$$\mathcal{E}[\![\nu]\!]\rho\delta\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\textbf{if}\ (\mathit{boolify}\ \varepsilon)\ \textbf{then}\ \mathcal{E}[\![\pi_1]\!]\ \textbf{else}\ \mathcal{E}[\![\pi_2]\!]\ \textbf{endif}\ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \to \varepsilon])\ \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\kappa\ \textbf{Значение}(\lambda\varepsilon^*\delta_1\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* = \#\nu^*\ \textbf{then}\ \mathit{allocate}\ \sigma_1\ \#\nu^*\ \lambda\sigma_2\alpha^*.(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \delta_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\varepsilon^*\sigma_2.(\varphi|_{\textbf{Функции}}\ \varepsilon^*\ \delta\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\ \S\ \varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\,]\!]\rho\delta\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\delta\kappa\sigma = \big(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma\big)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \kappa\ \sigma)$$

$$(\sigma_0\ (\rho_0\ [\![\texttt{call/cc}]\!])) =$$
$$\textbf{Значение}(\lambda\varepsilon^*\delta\kappa\sigma.\textbf{if}\ \#\varepsilon^* = 1\ \textbf{then}\ (\varepsilon^*{\downarrow_1}|_{\textbf{Функции}}\ \langle\textbf{Значение}(\lambda\varepsilon_1^*\delta_1\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon_1^* = 1\ \textbf{then}\ (\kappa\ \varepsilon_1^*{\downarrow_1}\ \sigma_1)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\rangle\ \delta\ \kappa\ \sigma)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})$$

Таблица 5.3. Scheme с динамическим окружением.

похожим образом, но поток может разделяться, например, при вычислении термов аппликации.

Динамические окружения принадлежат одноимённому домену, определяемому следующим образом:

$$\delta \quad \textbf{ДинамическоеОкружение} = \textbf{Переменная} \to \textbf{Значение}$$

Для работы с динамическим окружением предназначены две специальные формы: `dynamic-let` и `dynamic`. Форма `dynamic-let` устанавливает динамическую связь между *переменной* и *значением* на время вычисления *тела*. Она обрабатывает только одну переменную за раз, но это не проблема (макросы). Форма `dynamic` возвращает текущее значение динамической переменной; или ошибку, если запрошенная переменная не существует. Так как изменять динамические привязки мы запрещаем, то данное окружение связывает переменные со значениями напрямую, без посредников в виде адресов. Семантика рассмотренных форм показана в таблице 5.4, а вот их синтаксис:

`(dynamic-let (`*переменная значение*`)` *тело*`...)`
`(dynamic` *переменная*`)`

$$(\delta_0\ \nu) = \mathit{wrong}\ \texttt{"No such dynamic variable"}$$
$$\mathcal{E}[\![\texttt{(dynamic}\ \nu\texttt{)}]\!]\rho\delta\kappa\sigma = (\kappa\ (\delta\ \nu)\ \sigma)$$
$$\mathcal{E}[\![\texttt{(dynamic-let (}\nu\ \pi\texttt{)}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta[\nu \to \varepsilon]\ \kappa\ \sigma_1)\ \sigma)$$

Таблица 5.4. Семантика специальных форм динамического связывания.

Приведённая денотация довольно очевидна. Формы получают значение из динамического окружения или же расширяют его, таким образом мы получаем второе пространство имён для динамических переменных, отделённое от пространства лексических. И снова вы можете убедиться в огромной информативности подобной записи: лишь несколькими греческими буквами описывается новое пространство имён. Денотационная семантика одарит своей силой любого, кто сможет разобраться в её тайнописи.

Идея динамического окружения находит применение не только просто для переменных, но и при обработке ошибок, для реализации переходов и иных механизмов управления потоком исполнения [HD90, QD93]. Рассмотрим, например, простой, собранный на коленке механизм обработки ошибок: в случае неожиданной ситуации функция создаёт объект, описывающий, что именно произошло, и передаёт его функции, хранящейся в динамической переменной `*error*`. Теперь можно на любые вычисления навесить нужный нам обработчик ошибок, обернув эти вычисления в форму `dynamic-let`, устанавливающую необходимую функцию в `*error*`. Например, стандарт Scheme требует,

чтобы попытка открыть несуществующий файл вызывала ошибку, но, к сожалению, он не определяет функцию, позволяющую заранее проверить, существует ли файл. Используя динамические переменные, мы можем написать подобный предикат самостоятельно:

```
(define (file-exists? filename)
  (dynamic-let (*error* (lambda (anomaly) #f))
    (call-with-input-file filename
      (lambda (port) #t) ) ) )
```

Это приближённое определение, так как в действительности надо ещё проверить, что `*error*` вызвана потому, что файл действительно не существует, а не, например, потому что мы исчерпали лимит открытых портов ввода. Для этого необходимо знать структуру объектов, которые `call-with-input-file` может положить в `anomaly`.

5.7. Глобальное окружение

В этом разделе мы изучим глобальное окружение, рассмотрев несколько вариантов его денотации. Добавляется оно аналогично динамическому окружению δ. Как и локальное окружение ρ, глобальное окружение γ переводит идентификаторы в адреса. Глобальное окружение следует тенью за памятью: всякое вычисление теперь принимает, возможно использует и передаёт дальше не только память, но и глобальное окружение. В таблице 5.5 приведена денотация ядра Scheme с глобальным окружением. Из неё пока исключены ссылки на переменные и присваивание, их мы рассмотрим чуть позже.

5.7.1. Глобальное окружение в Scheme

С помощью подобного базиса можно реализовать несколько вариантов глобального окружения. В стандарте Scheme используется следующая вариация: 1) получить значение переменной можно только если она существует и инициализирована; 2) изменять значение переменной можно только если она существует; 3) переопределение переменной эквивалентно присваиванию.

Чтобы выразить первые два правила, нам необходим способ проверки, определена ли переменная в окружении. Для этого расширим область значений окружений: к домену адресов прибавим специальное значение, которое не является адресом и обозначает отсутствие привязки в окружении:

$$\begin{aligned} \textbf{ЛокальноеОкружение}: &\quad \textbf{Переменная} \to \textbf{Адрес} + \{\textit{no-binding}\} \\ \textbf{ГлобальноеОкружение}: &\quad \textbf{Переменная} \to \textbf{Адрес} + \{\textit{no-global-binding}\} \end{aligned}$$

Теперь смысл ссылок и присваиваний должен быть ясен: сначала мы ищем локальную переменную, если не находим, то продолжаем искать уже в глобальном окружении. В итоге у нас на руках или адрес, с которым мы идём

$$\mathcal{E}[\![\texttt{(if } \pi \ \pi_1 \ \pi_2\texttt{)}]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\textbf{if}\ (\mathit{boolify}\ \varepsilon)\ \textbf{then}\ \mathcal{E}[\![\pi_1]\!]\ \textbf{else}\ \mathcal{E}[\![\pi_2]\!]\ \textbf{endif}\ \rho\ \gamma_1\ \kappa\ \sigma_1)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{) }\pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma =$$
$$(\kappa\ \textbf{Значение}(\lambda\varepsilon^*\gamma_1\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* = \#\nu^*\ \textbf{then}\ \mathit{allocate}\ \sigma_1\ \#\nu^*\ \lambda\sigma_2\alpha^*.(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\ \gamma\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varphi\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma_1\ \lambda\varepsilon^*\gamma_2\sigma_2.(\varphi|_{\textbf{Функции}}\ \varepsilon^*\ \gamma_2\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma_1\ \lambda\varepsilon^*\gamma_2\sigma_2.(\kappa\ \langle\varepsilon\rangle\ \S\ \varepsilon^*\ \gamma_2\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\,]\!]\rho\gamma\kappa\sigma = (\kappa\ \langle\rangle\ \gamma\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin }\pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\gamma\kappa\sigma = \big(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)\ \sigma\big)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$(\sigma_0\ (\rho_0\ [\![\texttt{call/cc}]\!])) =$$
$$\textbf{Значение}(\lambda\varepsilon^*\gamma\kappa\sigma.\textbf{if}\ \#\varepsilon^* = 1\ \textbf{then}\ (\varepsilon^*\!\downarrow_1|_{\textbf{Функции}}\ \langle\textbf{Значение}(\lambda\varepsilon_1^*\gamma_1\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon_1^* = 1\ \textbf{then}\ (\kappa\ \varepsilon_1^*\!\downarrow_1\ \gamma_1\ \sigma_1)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})\rangle\ \gamma\ \kappa\ \sigma)\ \textbf{else}\ \mathit{wrong}\ \texttt{"Incorrect arity"}\ \textbf{endif})$$

Таблица 5.5. Scheme с глобальным окружением.

к памяти, или $\perp$, который отдаётся как есть. Конечно, кроме этого ещё надо подправить начальные окружения.

$$
\begin{array}{l}
(\rho_0\ \nu) = \textit{no-binding} \\
(\gamma_0\ \nu) = \textit{no-global-binding} \\
\mathcal{E}[\![\nu]\!]\rho\gamma\kappa\sigma = \textbf{let}\ \alpha = (\rho\ \nu) \\
\qquad\qquad \textbf{in if}\ \alpha = \textit{no-binding} \\
\qquad\qquad\quad \textbf{then let}\ \alpha_1 = (\gamma\ \nu) \\
\qquad\qquad\qquad\quad \textbf{in if}\ \alpha_1 = \textit{no-global-binding} \\
\qquad\qquad\qquad\qquad \textbf{then}\ \textit{wrong}\ \texttt{"No such variable"} \\
\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ (\sigma\ \alpha_1)\ \gamma\ \sigma) \\
\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\quad \textbf{else}\ (\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma) \\
\qquad\qquad\quad \textbf{endif} \\
\mathcal{E}[\![\texttt{(set!}\ \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\textbf{let}\ \alpha = (\rho\ \nu) \\
\qquad\qquad\qquad\qquad \textbf{in if}\ \alpha = \textit{no-binding} \\
\qquad\qquad\qquad\qquad\quad \textbf{then let}\ \alpha_1 = (\gamma_1\ \nu) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in if}\ \alpha_1 = \textit{no-global-binding} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \textit{wrong}\ \texttt{"No such variable"} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{endif}\ \sigma)
\end{array}
$$

Пусть глобальные переменные определяются специальной формой `define`, причём для простоты положим, что она (пере)определяет исключительно *глобальные* переменные (даже если находится внутри определения функции). Поэтому назовём её `define-global`. Итак, запишем её денотацию, которая должна или создать новую переменную, или изменить значение уже существующей:

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(define-global}\ \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\textbf{let}\ \alpha = (\gamma\ \nu) \\
\qquad\qquad\qquad\qquad \textbf{in if}\ \alpha = \textit{no-global-binding} \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \textit{allocate}\ \sigma_1\ 1\ \lambda\sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\kappa\ \varepsilon\ \gamma_1[\nu \to \alpha^*{\downarrow_1}]\ \sigma_2[\alpha^*{\downarrow_1} \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\quad \textbf{endif}\ \sigma)
\end{array}
$$

Единственная деталь, оставшаяся недоработанной, — это начальное состояние глобального окружения γ_0. Сейчас в нём нет ни одной переменной, но мы могли бы их туда добавить. Можно сделать так, чтобы в начальном глобальном окружении были лишь примитивы, а можно добавить туда все мыслимые переменные: раз уж все повторные определения переменной эквивалентны присваиванию ей, то пусть и первое будет таким же.

5.7.2. Автоматически расширяемое окружение

Некоторые диалекты Лиспа делают немного по-другому: в них первое присваивание переменной эквивалентно её определению. Это можно легко реализовать, вместо ошибки создавая новую переменную при присваивании.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(set! } \nu \ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\textbf{let}\ \alpha = (\rho\ \nu) \\
\qquad\qquad\qquad \textbf{in if}\ \alpha = \textit{no-binding} \\
\qquad\qquad\qquad\quad \textbf{then let}\ \alpha_1 = (\gamma_1\ \nu) \\
\qquad\qquad\qquad\qquad\quad \textbf{in if}\ \alpha_1 = \textit{no-global-binding} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \textit{allocate}\ \sigma_1\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda\sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\kappa\ \varepsilon\ \gamma_1[\nu \to \alpha^*{\downarrow_1}]\ \sigma_2[\alpha^*{\downarrow_1} \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \to \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad\quad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon]) \\
\qquad\qquad\qquad\quad \textbf{endif}\ \sigma)
\end{array}
$$

Преимущество такой вариации в том, что нам больше не требуется отдельная форма `define`. Недостаток же в том, что ошибки в именах переменных становятся менее заметными, так как никаких предупреждений о неопределённых переменных уже не выводится.

5.7.3. Гиперстатическое окружение

В гиперстатическом случае замыкания захватывают не только локальное окружение, но и текущее состояние глобального. Это поведение реализуется простым изменением определения абстракции из таблицы 5.5:

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{) }\pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
\quad (\kappa\ \textbf{Значение}(\lambda\varepsilon^*\gamma_1\kappa_1\sigma_1.\textbf{if}\ \#\varepsilon^* = \#\nu^* \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \textit{allocate}\ \sigma_1\ \#\nu^* \\
\qquad\qquad\qquad\qquad\qquad\quad \lambda\sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ \textit{wrong}\ \texttt{"Incorrect arity"} \\
\qquad\qquad\qquad\qquad\quad \textbf{endif})\ \gamma\ \sigma)
\end{array}
$$

Если присваивание, как в Scheme, может изменять значения только существующих переменных, то такое определение не вызывает проблем. Если же `set!` может работать как `define`, то поведение уже не всегда очевидно. Например:

```
(define (weird v)
  (set! a-new-variable v) )
```

Присваивание внутри `weird` расширяет глобальное окружение, захваченное замыканием. Но состояние этого окружения не сохраняется между вызовами `weird`, каждый из них начинает с чистого листа.

Также, как мы упоминали ранее, гиперстатическое глобальное окружение не дружит со взаимно рекурсивными функциями. Можно было бы ввести специальную форму для множественных одновременных определений, но лучше создать новый механизм для доступа к глобальным переменным: `(global` ν`)`. Эта специальная форма позволяет обратиться к глобальной переменной ν в *любом* контексте, даже если существует одноимённая локальная переменная. Следовательно, мы сможем определять глобальные взаимно рекурсивные функции следующим образом:

```
(letrec ((odd? (lambda (n) (if (= n 0) #f (even? (- n 1)))))
         (even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) )
  (define-global odd? odd?)
  (define-global even? even?) )
```

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(global } \nu\texttt{)}]\!]\rho\gamma\kappa\sigma = \textbf{let } \alpha = (\gamma\ \nu)\\
&\qquad\qquad\qquad\qquad \textbf{in if } \alpha = \textit{no-global-binding}\\
&\qquad\qquad\qquad\qquad\quad \textbf{then } \textit{wrong } \texttt{"No such variable"}\\
&\qquad\qquad\qquad\qquad\quad \textbf{else } (\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma)\\
&\qquad\qquad\qquad\qquad\quad \textbf{endif}\\
&\mathcal{E}[\![\texttt{(define-global } \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma =\\
&\quad (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\textbf{let } \alpha = (\gamma\ \nu)\\
&\qquad\qquad\qquad\qquad \textbf{in if } \alpha = \textit{no-global-binding}\\
&\qquad\qquad\qquad\qquad\quad \textbf{then } \textit{allocate } \sigma_1\ 1\\
&\qquad\qquad\qquad\qquad\qquad\qquad \lambda\sigma_2\alpha^*.\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\kappa\ \varepsilon\ \gamma_1[\nu \to \alpha^*{\downarrow_1}]\ \sigma_2[\alpha^*{\downarrow_1} \to \varepsilon])\\
&\qquad\qquad\qquad\qquad\quad \textbf{else } (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \to \varepsilon])\\
&\qquad\qquad\qquad\qquad\quad \textbf{endif } \sigma)
\end{aligned}$$

И снова мы видим, как денотационная семантика позволяет элегантно описывать разнообразные варианты окружений. Конечно, мы также могли бы легко объединить определённые выше окружения, получив множественные пространства имён, как это сделано в Common Lisp.

5.8. Под капотом у денотаций

Предназначением данной главы было снять покров таинственности с денотационной семантики. Здесь это сделано весьма неформально (кое-кто бы поправил: „кощунственно“), так как именно такой подход полезен для популяризации денотационной семантики. Мы постепенно уточняли определяемый язык с помощью различных интерпретаторов, одновременно с этим используя всё более и более ограниченный язык описания; в итоге к этой главе мы дошли до чистой математики, до денотационного интерпретатора.

Scheme и λ-исчисление — дальние родственники, но всё же родственники. Поэтому вполне возможно сделать денотации исполнимыми, чтобы получить возможность исправить ошибки в этих запутанных уравнениях. Возможность перепроверить правильность денотаций с помощью компьютера очень важна: так можно убедиться в том, что язык ведёт себя именно так, как задумано; это позволяет экспериментировать с языком без опаски что-либо сломать. Так как денотации почти непосредственно исполнимы, отсутствует необходимость писать сложный интерпретатор и доказывать, что он правильно понимает их семантику.

Поэтому писать транслятор денотаций в исполнимый код приятно и полезно. Он не уменьшает мощь λ-исчисления, хоть и накладывает одно ограничение: все вызовы производятся по значению (используется «плохое» правило вычислений Scheme). Но эта не такая уж и проблема. Заявляю при свидетелях: все денотации, приведённые в данной главе, на самом деле выполнены на Scheme и пропущены через небольшой препроцессор (LISP2TEX), который перевёл их на греческий [Que93d]. Представьте себе, скольких трудов стоило бы сделать денотации исполнимыми (и протестировать их), если бы для симуляции аппликативного λ-исчисления (в котором, к тому же, ещё и порядок вычислений должен быть неопределённым) использовался бы не аппликативный язык!

Вот пример исходного кода для денотации простой абстракции (с фиксированной арностью). Можете сравнить его с соответствующим «греческим профилем» на странице 197.

```
(define ((meaning-abstraction n* e+) r k s)
  (k (inValue (lambda (v* k1 s1)
                (if (= (length v*) (length n*))
                    (allocate s1 (length n*)
                              (lambda (s2 a*)
                                ((meaning*-sequence e+)
                                 (extend* r n a*)
                                 k1
                                 (extend* s2 a* v*) ) ) )
                    (wrong "Incorrect arity") ) ))
      s ) )
```

Здесь используется устаревшая форма `define`, которая понимает `(define (`ν `.` *переменные*`)` π^*`)` как `(define` ν `(lambda` *переменные* π^*`))`, где ν описывает форму вызова определяемой функции.

Заданные подобным образом функции ставятся под начало синтаксического анализатора, который по получаемому выражению определяет соответствующую функцию для его обработки. Его структура вам давно знакома:

```
(define (meaning e)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e)
                      (meaning-quotation e) )
```

```
(case (car e)
  ((quote)  (meaning-quotatione (cadr e)))
  ((lambda) (meaning-abstraction (cadr e) (cddr e)))
  ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)))
  ((begin)  (meaning-sequence (cdr e)))
  ((set!)   (meaning-assigment (cadr e) (caddr e)))
  (else     (meaning-application (car e) (cdr e))) ) ) )
```

5.9. λ-исчисление и Scheme

Так как для нас важен вопрос исполнимости денотаций, то следует подробнее разобраться в различиях между Scheme и λ-исчислением. (Естественно, имеется в виду «целомудренное» подмножество Scheme, не испорченное побочными эффектами, присваиваниями и т. п.) Самое главное отличие лежит в порядке вычислений. В λ-исчислении нет фиксированного порядка, есть только не очень хорошие порядки, которых стоит избегать. В Scheme же всё по-другому: здесь обязателен аппликативный порядок вычислений. То есть, хоть последовательность вычисления аргументов и не определена, но все они должны быть вычислены до применения функции. Кроме того, это же правило запрещает вычислять тела `lambda`-форм раньше времени.

Вызовы по имени можно проэмулировать в Scheme с помощью *обещаний* (promises, также известны как thunks, futures или delays). Обещание — это замыкание без параметров, инкапсулирующее отложенные вычисления. Мы можем дать обещание что-то вычислить с помощью `delay` и потребовать выполнения обещания с помощью `force`. Эти функции на Scheme определяются элементарно:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression) (lambda () expression)) ) )

(define (force promise) (promise))
```

Форма `delay` замыкает вычисления вместе с их окружением в обещании, которое можно выполнить, передав его `force`. Используя обещания, можно легко проэмулировать вызов по имени, преобразуя каждый вызов (`f` a `...` z) в (`f` (`delay` a) `...` (`delay` z)), а когда нам внутри понадобится значение аргумента, мы его затребуем с помощью `force`. Рассмотрим пример. Вот выражение, с которым у нас были проблемы из-за аппликативного порядка вычислений:

```
(((lambda (x) (lambda (y) y))                    ; ((λx.λy.y (ω ω)) z)
  ((lambda (x) (x x)) (lambda (x) (x x))) ) z )
```

Использовав вышеуказанный приём, мы задерживаем вычисление $(\omega\ \omega)$ до тех пор, пока его значение не потребуется (то есть навсегда), и возвра-

щаем правильное значение нормальной формы этого выражения — значение свободной переменной `z`.

```
(((lambda (x) (lambda (y) (delay y)))
  (delay ((lambda (x) ((force x) (delay (force x))))
          (lambda (x) ((force x) (delay (force x)))) )) )
 (delay z) )
```

Хотя это несомненно работает [DH92], но вычисления выполняются неэффективно. Не говоря уже о бессмысленных `(delay (force x))`, сейчас каждый вызов `force` требует вычислить нам значение *заново*. А ведь мы можем вычислить его один раз, сохранить где-нибудь и впоследствии просто возвращать уже сохранённое значение. Такой приём называется вызовом по необходимости или *мемоизацией*. Для его использования потребуется изменить определение `delay`, чтобы обещание при его выполнении запоминало результат. К счастью, Scheme прекрасно подходит для метапрограммирования, так что реализация такого поведения не представляет труда:

```
(define-syntax memo-delay
  (synatax-rules ()
    ((memo-delay expression)
     (let ((already-computed? #f)
           (value 'wait) )
       (lambda ()
         (if (not already-computed?)
             (begin (set! value expression)
                    (set! already-computed? #t) ) )
         value ) ) ) ) )
```

Конечно, вовсе необязательно преобразовывать все вызовы вручную, когда в языке есть макросы, см. [DFH86]. Если же мы желаем максимальной эффективности, то существует техника *анализа строгости* (strictness analysis), которая определяет, какие объекты бессмысленно заворачивать в обещания, потому что они вычисляются всегда [BHY88]. Наконец, сами обещания можно представлять иным образом, чтобы минимизировать затраты времени на проверку `(if (not already-computed?) ...)`. Обещания добавляют в Scheme ленивые вычисления, где преобразования лишь описываются, а выполняются уже самостоятельно и автоматически в нужное время. Однако такой стиль программирования вызывает очевидные затруднения при отладке, а также не особо сочетается с побочными эффектами и продолжениями. Сравним, например, следующие программы из [KW90, Mor92], они отличаются лишь на одну `delay`, но дают совершенно различные результаты:

```
(pair? (call/cc (lambda (k) (list (k 33)))))
(pair? (call/cc (lambda (k) (list (delay (k 33))))))
```

5.9.1. Круговорот продолжений в природе

Только совсем уж невнимательный читатель ещё не заметил, что все денотации были записаны в *стиле передачи продолжений* (continuation passing style). Очевидно, что мы можем транслировать любую программу на Scheme, использует она `call/cc` или нет, в её денотационный эквивалент в λ-исчислении. Но ведь сами λ-термы тоже однозначно представляются в виде программ на Scheme, которые не используют `call/cc`. Вопрос: а можно ли преобразовать программу на Scheme, содержащую `call/cc`, в эквивалентную программу на Scheme, но без `call/cc`?

Ответ: да, но в этом случае нам придётся передавать продолжения явно. К счастью, с этим нет особых проблем, так как продолжения — это те же `lambda`-формы. В действительности, некоторые компиляторы [App92a] намеренно переводят компилируемые программы сначала в такой промежуточный вид, чтобы избавиться от «особенной» формы `call/cc`. Однако, у этого приёма есть и недостатки: он сильно вредит читабельности получаемого кода, а также иногда преждевременно вносит упорядоченность в вычисления. Тем не менее, программы, как показано в [SF92], действительно остаются эквивалентными. Преобразование, рассматриваемое здесь, вдохновлено работой [DF90]. В разделе 10.11.2 рассматривается иной вариант. [**см. стр. 480**]

Перейдём к реализации. Мы полагаем, что отныне всякая функция принимает дополнительный аргумент[5] — своё продолжение. То есть $_k$`(foo bar ... hux)` превращается в `(foo` k `bar ... hux)`. С продолжениями разобрались, с функциями тоже, остались специальные формы. Их разбор и преобразование производится обычным способом. Продолжения в них используются точно так же, как и в денотациях: для вычисления и хранения промежуточных результатов, а также для передачи управления следующему коду (оставшимся вычислениям).

```
(define (cps e)
  (if (atom? e) (lambda (k) (k ‘,e))
      (case (car e)
        ((quote)  (cps-quote (cadr e)))
        ((if)     (cps-if (cadr e) (caddr e) (cadddr e)))
        ((begin)  (cps-begin (cdr e)))
        ((set!)   (cps-set! (cadr e) (caddr e)))
        ((lambda) (cps-abstraction (cadr e) (cddr e)))
        (else     (cps-application e)) ) ) )
```

Транслятор `cps` принимает программу, выполняет преобразования и возвращает замыкание, являющееся той же программой в стиле передачи продолжений. Такая программа принимает своё продолжение (такое же замыкание) и возвращает результат вычислений, представляемый ещё одним замыканием.

[5] Он передаётся первым, чтобы упростить работу с функциями переменной арности.

В итоге, `cps` имеет следующий своеобразный тип:

$$\textbf{Программа} \to \big((\textbf{Программа} \to \textbf{Программа}) \to \textbf{Программа}\big)$$

Цитирование снова элементарно:

```
(define (cps-quote data)
  (lambda (k)
    (k `(quote ,data)) ) )
```

Присваивание тоже просто записывается:

```
(define (cps-set! variable form)
  (lambda (k)
    ((cps form)
     (lambda (a)
       (k `(set! ,variable ,a)) ) ) ) )
```

Мы вычисляем новое значение переменной `(cps form)`, передаём его промежуточному продолжению через переменную `a`, затем передаём результат присваивания дальнейшей программе.

Условный оператор действует аналогично:

```
(define (cps-if bool formT formF)
  (lambda (k)
    ((cps bool)
     (lambda (b)
       `(if ,b ,((cps formT) k)
               ,((cps formF) k) ) ) ) ) )
```

Упорядочить вычисления чуть сложнее:

```
(define (cps-begin e)
  (if (pair? e)
      (if (pair? (cdr e))
          (let ((void (gensym "void")))
            (lambda (k)
              ((cps (car e))
               (lambda (a)
                 ((cps-begin (cdr e))
                  (lambda (b)
                    (k `((lambda (,void) ,b) ,a)) ) ) ) ) ) )
          (cps (car e)) )
      (cps '()) ) )
```

Здесь интересным местом является способ игнорирования промежуточных значений.

Самое сложное — это обработка `lambda`-форм. Им всем надо добавить ещё один аргумент — продолжение — и гарантировать правильность его передачи.

Кроме этого мы введём небольшую оптимизацию для простых функций (которые быстро выполняются и всегда возвращают результат). Список `primitives` определяет перечень таких функций.[6]

```
(define (cps-application e)
  (lambda (k)
    (if (memq (car e) primitives)
        ((cps-terms (cdr e))
         (lambda (t*)
           (k `(,(car e) ,@t*)) ) )
        ((cps-terms e)
         (lambda (t*)
           (let ((d (gensym)))
             `(,(car t*) (lambda (,d) ,(k d))
                         . ,(cdr t*) ) ) ) ) ) ) )

(define primitives '(cons car cdr list * + - = pair? eq?))

(define (cps-terms e*)
  (if (pair? e*)
      (lambda (k)
        ((cps (car e*))
         (lambda (a)
           ((cps-terms (cdr e*))
            (lambda (a*)
              (k (cons a a*)) ) ) ) ) )
      (lambda (k) (k '())) ) )

(define (cps-abstraction variables body)
  (lambda (k)
    (k (let ((c (gensym "cont")))
         `(lambda (,c . ,variables)
            ,((cps `(begin ,@body))
              (lambda (a) `(,c ,a)) ) ) )) ) )
```

Готово. Посмотрим, во что данная трансформация превратит факториал:

```
(set! fact (lambda (n)
             (if (= n 1) 1
                 (* n (fact (- n 1))) ) ))
⇝ (set! fact
        (lambda (cont112 n)
          (if (= n 1)
              (cont112 1)
              (fact (lambda (g113) (cont112 (* n g113)))
                    (- n 1) ) ) ) )
```

[6] Правильная оптимизация вызовов предопределённых примитивов рассматривается в разделе 6.1.8.

Теперь мы автоматически получаем то, что раньше были вынуждены писать вручную. Заметьте, что преобразование превратило программу в последовательность простых вызовов функций: сравнений, арифметики и продолжений. Нет никаких дополнительных особых случаев, кроме специальных форм.

После CPS-преобразования форма $_k$`(call/cc f)` превращается в `(call/cc` k `f)`, а сама функция `call/cc` становится вообще тривиальной: `(lambda (k f) (f k k))`, поэтому мы на ней и не останавливались. Единственная загвоздка в том, чтобы сами продолжения, возвращаемые `call/cc`, оставались функциями. То есть, чтобы форма `(procedure? (apply call/cc (list call/cc)))` возвращала истину.

Одним из достоинств стиля передачи продолжений является то, что благодаря явному указанию, когда что вычислять и кому возвращать результат, становится без разницы, нормальный ли порядок принят в языке реализации или аппликативный. В обоих случаях мы получим одинаковое поведение. Поэтому использование обещаний совместно с подобным стилем позволяет сократить разрыв между Scheme и λ-исчислением [DH92].

5.9.2. Динамическое окружение

В предыдущем разделе денотации натолкнули нас на идею преобразования программ, позволяющего избавиться от `call/cc`. Ранее мы рассматривали динамическое окружение; а ведь мы можем аналогичным образом избавиться и от форм `dynamic-let` и `dynamic`. Для этого потребуется явно ввести динамическое окружение и определить соответствующее преобразование программ.

Предположим, у нас есть идентификатор, который не используется ни в одной программе. Назовём его δ. Свяжем его с динамическим окружением — функцией, которая по имени динамической переменной возвращает её значение. Предположим, что у нас есть функция `update`, которая умеет расширять подобное окружение [**см. стр. 163**], которая доступна отовсюду, которую нельзя переопределить, перекрыть локальными переменными и т. п. Используя всё это, можно реализовать преобразования $\mathcal{D}$ и $\mathcal{D}^*$, приведённые в таблице 5.6.

$\mathcal{D}^*[\![\,]\!]$	$\to$	
$\mathcal{D}^*[\![\pi\ \pi^*]\!]$	$\to$	$\mathcal{D}[\![\pi]\!]\ \mathcal{D}^*[\![\pi^*]\!]$
$\mathcal{D}[\![$`(if` $\pi_c\ \pi_t\ \pi_f$`)`$]\!]$	$\to$	`(if` $\mathcal{D}[\![\pi_c]\!]\ \mathcal{D}[\![\pi_t]\!]\ \mathcal{D}[\![\pi_f]\!]$`)`
$\mathcal{D}[\![$`(begin` π^*`)`$]\!]$	$\to$	`(begin` $\mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(`$\pi\ \pi^*$`)`$]\!]$	$\to$	`(`$\mathcal{D}[\![\pi]\!]\ \delta\ \mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(lambda (`ν^*`)` π^*`)`$]\!]$	$\to$	`(lambda (`$\delta\ \nu^*$`)` $\mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(dynamic` ν`)`$]\!]$	$\to$	`(`δ `(quote` ν`))`
$\mathcal{D}[\![$`(dynamic-let (`$\nu\ \pi$`)` π^+`)`$]\!]$	$\to$	`(let ((`δ `(update` δ `(quote` ν`)` π`)))` $\mathcal{D}^*[\![\pi^+]\!]$`)`

Таблица 5.6. Трансформация, убирающая динамическое окружение.

Таким образом мы можем проэмулировать динамическое окружение, если не можем или не хотим встраивать его поддержку в ядро языка. Последний штрих: начальное динамическое окружение. Программа π преобразуется в

`(let ((δ (lambda (n) (error "No such dynamic variable" n)))) 𝒟⟦π⟧)`

В итоге можно сделать следующий вывод: денотации специальных форм языка иногда способны показать, что некоторые формы вовсе не такие уж и специальные.

5.10. Заключение

Данная глава венчает серию интерпретаторов, определяющих Scheme всё точнее и точнее с использованием всё более ограниченных средств. Денотационная семантика, по крайней мере в рассмотренном виде, позволяет очень точно и лаконично описывать *ядро* языка. Она плохо подходит для описания всего языка, его мельчайших деталей, так как на таком уровне нотация уже чрезмерно усложняется.

В данной главе среди важных вещей мы не рассмотрели сравнение функций, денотацию констант, а также всевозможные не особо важные тонкости, которые редко заметны и ещё реже используются. Денотационная семантика прекрасно подходит для набрасывания общей формы языка, но проработка деталей с её помощью становится неимоверно скучной.

Существует огромное множество вещей, которые можно описать с помощью денотационной семантики. Например, организовать параллелизм с помощью техники пошаговых вычислений [Que90c]. Или распределённое хранение и обработку данных [Que92b]. Но есть также вещи, которые таким образом описывать неудобно [McD93]. К примеру, вывод типов довольно сложно выразить денотационно, именно поэтому существует естественная семантика [Kah87].

5.11. Упражнения

Упражнение 5.1 Рассмотрим ещё один способ денотации аппликации. Докажите, что он эквивалентен показанному в разделе 5.2.6.

$$\mathcal{E}[\![(\pi\ \ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\overline{\mathcal{E}}[\![\pi^*]\!]\ \langle\rangle\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi|_{\textbf{Функции}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\overline{\mathcal{E}}[\![]\!]\varepsilon^*\rho\kappa\sigma = (\kappa\ (\mathit{reverse}\ \varepsilon^*)\ \sigma)$$

$$\overline{\mathcal{E}}[\![\pi\ \pi^*]\!]\varepsilon^*\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\overline{\mathcal{E}}[\![\pi^*]\!]\ \langle\varepsilon\rangle \S \varepsilon^*\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

Упражнение 5.2 Определения из раздела 5.3 слишком затрудняют описание рекурсивных функций. Мы могли бы, как в Lisp 1.5, ввести специальную форму `label`. В Scheme форма `(label` ν `(lambda ...))` эквивалентна `(letrec ((`ν `(lambda ...))) `ν`)`. Определите семантику аналогичного оператора `label` для λ-исчисления.

Упражнение 5.3 Измените денотацию `dynamic` таким образом, чтобы в случае отсутствия динамической переменной с искомым именем возвращалась одноимённая переменная из глобального окружения.

Упражнение 5.4 Напишите макрос, который бы эмулировал неопределённый порядок вычисления аргументов в такой реализации Scheme, где аргументы вычисляются слева направо. В вашем распоряжении есть унарная функция `random-permutation`, которая принимает целое число n и возвращает случайную перестановку чисел $0, \ldots, n-1$.

Рекомендуемая литература

Я настойчиво рекомендую обратить внимание на книги [Sto77] и [Sch86]. В обеих содержатся просто горы информации о денотационной семантике, а также примеры денотирования языков.

Тем же, кого серьёзно зацепило λ-исчисление, стоит приняться за классическую книгу [Bar84].

Глава 6

Быстрая интерпретация

Денотационный интерпретатор из предыдущей главы чрезвычайно точно определяет язык, но работает невероятно медленно. В этой главе мы разберём причины его неторопливости и разработаем несколько новых интерпретаторов, исправляющих данный недостаток с помощью предобработки программ. Короче говоря, в этой главе будет создан зачаточный компилятор. В процессе мы проанализируем представление лексических окружений, соглашения вызова функций и реализацию продолжений. Предварительная обработка программ будет заключаться в нахождении и устранении вычислений, которые могут быть выполнены заранее, ещё до начала исполнения программы; в результате в наших программах останутся только необходимые вычисления. Эти вычисления будут представляться специальными комбинаторами, играющими роль промежуточного языка, похожего на язык ассемблера (высокоуровневой) виртуальной машины.

В предыдущей главе мы добились восхитительной точности работы интерпретатора. Теперь пришло время исправить его отвратительную медлительность. Естественно, мы не будем рубить с плеча, а продолжим неспешно и тщательно дорабатывать существующие решения. Отчасти потому, что денотационный интерпретатор установил планку точности описания языка, ниже которой опускаться не следует. Поэтому мы рассмотрим один за другим три интерпретатора, постепенно принося в жертву некоторые их возможности ради удобства и простоты реализации.

6.1. Быстрый интерпретатор

Ради эффективности предположим, что язык реализации содержит некоторый минимальный набор средств, в частности, память. Первой на покой уйдёт наша инкрементальная память из четвёртой главы. [**см. стр. 142**] Она была введена исключительно как иллюстрация концепции, а не в роли практической рекомендации. Конечно, связывание при этом никуда не денется, но память будет использоваться лишь для хранения привязок. Теперь это проблемы `cons`, где и как размещать содержимое точечных пар, то же самое касается `vector`, `lambda` и прочих. Память интерпретатора больше не будет

жуткой свалкой из аргументов функций, кусков структур данных, замыканий, меток переходов и т. п. Отныне там будут лежать исключительно привязки, упакованные в записи активаций.

6.1.1. Миграция денотаций

Одной из главных причин неэффективности нашего интерпретатора является чрезмерное использование абстракций для детализации конструкций языка, что вкупе с подходом Scheme к вычислению аргументов приводит к огромной доле вычислений, выполняемых в неподходящее время. Сейчас нас устраивает достигнутый уровень точности, так что можно перенести вычисления немного вперёд, чтобы выполнять их раньше [Deu89]. Такой перенос называется *миграцией* кода, также известной под именами λ-*hoisting* [Tak88] и λ-*drifting* [Roz92]. Конечно, в Scheme его надо применять осторожно, чтобы не вызвать какие-либо побочные эффекты в неправильное время, а то и неправильное количество раз. Также при переносе могут возникнуть проблемы с бесконечными циклами. Действительно, выражение `(lambda (x) (`ω ω`))`, где вычисление `(`ω ω`)` никогда не закончится, не является эквивалентом `(let ((tmp (`ω ω`))) (lambda (x) tmp))`. Однако, обе эти проблемы принципиально отсутствуют в денотационном интерпретаторе, так как в λ-исчислении нет побочных эффектов, а те денотации, которые имеет смысл переносить, являются композициональными и получаются из программ конечных размеров, так что они гарантированно вычислимы за конечное время.

В качестве примера рассмотрим новую версию абстракции. Денотация тела функции `(meaning*-sequence e+)` и количество её аргументов `(length n*)` теперь определяются лишь однажды:

```
(define (meaning-abstraction n* e+)
  (let ((m (meaning*-sequence e+))
        (arity (length n*)) )
    (lambda (r k s)
      (k (inValue (lambda (v* k1 s1)
                    (if (= (length v*) arity)
                        (allocate s1 arity
                                  (lambda (s2 a*)
                                    (m (extend* r n* a*)
                                       k1
                                       (extend* s2 a* v*) ) ) )
                        (wrong "Incorrect arity") ) ))
         s ) ) ) )
```

6.1.2. Записи активаций

Очевидно, что оптимизировав потребление памяти, мы сможем существенно повысить скорость интерпретации. Одной из наиболее прожорливых частей

денотационного интерпретатора является протокол вызова функций. Имеется в виду механизм, с помощью которого функции получают доступ к своим аргументам. В денотационном интерпретаторе аргументы функции представляются списками, а окружения — списками списков аргументов. Единственное оправдание настолько наивного механизма передачи аргументов — это то, что мы разрабатывали прототип. Для серьёзного, быстрого интерпретатора такое нагромождение списков неприемлемо.

Некоторые выделения памяти явно излишни, но есть и неизбежные расходы. Для разработчика естественно желание избавиться от первых за счёт вторых. При вызове функции нам обязательно надо передать ей аргументы, так что нельзя не выделять память под них: это могут быть занятые регистры, кусочек стека или же *запись активации*. Запись активации функции — это некий объект, хранящий конкретный набор аргументов, переданных функции при вызове (при её активации). Чаще всего эти записи фактически располагаются в аппаратном стеке, поэтому их также называют стековыми кадрами (фреймами). Пока что представим, что они определяются так:

```
(define-class activation-frame Object                    Временно
  ( (* argument) ) )
```

В этом определении используется новая возможность нашей объектной системы: звёздочка `*` обозначает *индексированное* поле; такие поля содержат не одно значение, а упорядоченный набор значений, размер которого определяется при создании объекта. (См. раздел 11.2 за подробностями.)

Такое представление эффективнее расходует память, когда аргументов больше двух. Также оно позволяет получить прямой доступ к элементам, тогда как в списке время доступа к произвольному элементу линейно зависит от размера списка. Конечно, введение записей активаций не должно мешать нам расширять окружения. Можно, например, представлять окружения списками записей активаций, как на рисунке 6.1.

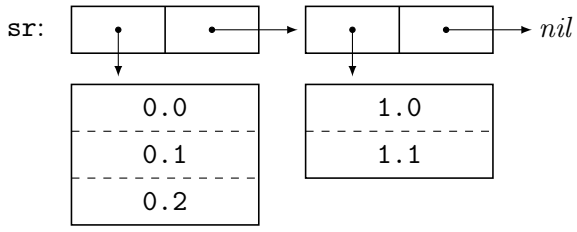


Рис. 6.1. Окружение как список записей активаций.

Можно сделать ещё лучше, введя специальное поле, с помощью которого записи активаций будут связываться в список. Так как время на выделение памяти в среднем слабо зависит от размера объекта (если закрыть глаза на инициализацию), то выделить память для одной записи активации с лишним полем выходит быстрее, чем для записи активации и точечной пары отдельно.

Так и поступим (рисунок 6.2), сделав подобное поле первым, чтобы его расположение не зависело от количества хранимых в записи переменных.

```
(define-class environment Object
  ( next ) )

(define-class activation-frame environment
  ( (* argument) ) )
```

Значения, образующие лексическое окружение, хранятся в связном списке записей активаций. Функция `sr-extend*` позволяет добавить в окружение новую запись:

```
(define (sr-extend* sr v*)
  (set-environment-next! v* sr)
  v* )
```

Как видите, записи активаций физически модифицируются при расширении окружения, так что функции не могут разделять их между собой. Каждый вызов функции создаёт собственные привязки, а значит, новую запись активации.

Теперь вспомним, как представляются лексические окружения в денотационном интерпретаторе: они состоят из двух отдельных частей: ρ и σ. Окружение ρ связывает имена переменных с адресами их значений в памяти σ. Но у нас сейчас практически не осталось памяти как таковой. Адресом переменной является её положение в цепочке записей активаций, которое описывается двумя числами: номером записи активации и индексом переменной в ней. Следующие две функции определяют чтение и модификацию таких переменных.

```
(define (deep-fetch sr i j)
  (if (= i 0)
      (activation-frame-argument sr j)
      (deep-fetch (environment-next sr) (- i 1) j) ) )

(define (deep-update! sr i j v)
  (if (= i 0)
      (set-activation-frame-argument! sr j v)
      (deep-update! (environment-next sr) (- i 1) j v) ) )
```

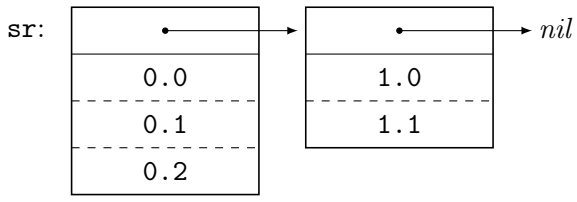


Рис. 6.2. Окружение как непосредственно связный список.

Записи активаций представляют связи между адресами и значениями, а окружения — между именами и адресами переменных. Поэтом теперь память (от прямой манипуляции которой мы хотим избавиться) фактически хранит исключительно записи активаций. Окружения по аналогии с ρ мы будем называть `r`. Память является *состоянием* всех окружений, поэтому она именуется `sr`. Сами окружения, по-хорошему, стоило бы представлять подобно памяти, но знакомые вам «гирлянды» из списков [**см. упр. 1.3**] являются вполне приемлемым вариантом. Окружения расширяются функцией `r-extend*` (не путать с `sr-extend*`); кроме неё нам потребуется «полупредикат» `local-variable?`, возвращающий *лексические индексы* переменных.

```
(define (r-extend* r n*)
  (cons n* r) )

(define (local-variable? r i n)
  (and (pair? r)
       (let scan ((names (car r))
                  (j 0) )
         (cond ((pair? names)
                (if (eq? n (car names))
                    `(local ,i . ,j)
                    (scan (cdr names) (+ 1 j)) ) )
               ((null? names)
                (local-variable? (cdr r) (+ i 1) n) )
               ((eq? n names) `(local ,i . ,j)) ) ) ) )
```

Всё это сделано для того, чтобы привести структуру памяти к строго иерархическому виду. Сейчас адрес переменной — это не абсолютный номер ячейки памяти, а пара *относительных* координат в текущем списке записей активаций. Таким образом, сами адреса являются статическими и могут мигрировать на этапе предварительной обработки, а значения переменных определяются как обычно: динамически с помощью адресов и текущего состояния памяти.

Подобное разделение лексического окружения на статическую и динамическую части не зависит от реализации памяти. На самом деле, это глубинное свойство самих лексических окружений, которое стало явным только сейчас. Реализация памяти в денотационном интерпретаторе тоже позволяет предсказывать «положение» значения переменной в памяти (считая положением количество сравнений адресов при поиске) по окружению, так как окружения представляются цепочками замыканий вида `(lambda (u) (if (eq? u` x`)` y `(`f `u)))`.

6.1.3. Интерпретатор: начало

Сейчас мы уже знаем достаточно для того, чтобы набросать ядро интерпретатора и реализовать несколько специальных форм. Ожидаемо, начнём

мы работу с функции `meaning`, имеющей следующий тип:

$$\texttt{meaning}: \underbrace{\textbf{Программа} \times \textbf{Окружение}}_{\text{статическая часть}} \rightarrow \rightarrow (\underbrace{\textbf{ЗаписиАктиваций}^* \times \textbf{Продолжение} \rightarrow \textbf{Значение}}_{\text{динамическая часть}})$$

Здесь чётко видно, как именно лексические окружения разделяются на две части: статические привязки имён к адресам в **Окружениях** и динамические привязки адресов к значениям в **ЗаписяхАктиваций**. Программа, прошедшая подобную *предобработку*, превращается в функцию, ожидающую связный список записей активаций (состояние памяти) и продолжение для того, чтобы вычислить результат. Такой необычный способ представления программ весьма далёк от изначальных S-выражений, но принципиально остаётся тем же.

Как обычно, ради упрощения синтаксического анализа разбираемые программы считаются синтаксически корректными. Вспомним наши соглашения именования сущностей:

`e`	выражения, формы
`r`	окружения
`sr`, `v*`	записи активаций
`k`, `kk`	продолжения
`v`	значения
`f`	функции
`n`	идентификаторы

Отлично, вот синтаксический анализатор:

```
(define (meaning e r)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r)
                      (meaning-quotation e r) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e) r))
        ((begin)  (meaning-sequence (cdr e) r))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r))
        (else     (meaning-application (car e) (cdr e) r)) ) ) )
```

Цитирование в этот раз тривиально:

```
(define (meaning-quotation v r)
  (lambda (sr k)
    (k v) ) )
```

Ветвление показывает хороший пример миграции кода. Предварительную обработку проходят обе ветки независимо от того, какая из них будет выбрана в действительности.

```
(define (meaning-alternative e1 e2 e3 r)
  (let ((m1 (meaning e1 r))
        (m2 (meaning e2 r))
        (m3 (meaning e3 r)) )
    (lambda (sr k)
      (m1 sr (lambda (v)
               ((if v m2 m3) sr k)) ) ) ) )
```

Упорядочиваются формы как обычно, с помощью рассмотрения двух возможных вариантов списка форм.

```
(define (meaning-sequence e+ r)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r)
          (meaning*-single-sequence (car e+) r) )
      (static-wrong "Illegal syntax: (begin)") ) )

(define (meaning*-single-sequence e r)
  (meaning e r) )

(define (meaning*-multiple-sequence e e+ r)
  (let ((m1 (meaning e r))
        (m+ (meaning-sequence e+ r)) )
    (lambda (sr k)
      (m1 sr (lambda (v)
               (m+ sr k)) ) ) ) )
```

Сложности начинаются с аппликацией. Для неё нам придётся точнее описать протокол вызова функций.

Аппликация

Для обработки аппликаций необходимо определить несколько вещей: как выделяется память под аргументы, как она заполняется и как передаётся, — короче говоря, как используются записи активаций. Предварительная обработка функционального терма аппликации проблем не представляет, аргументы требуют чуть больше внимания, так что для их обработки мы введём функцию `meaning*`. Наконец, тела функций для простоты реализуются с помощью родных замыканий Scheme.

```
(define (meaning-regular-application e e* r)
  (let* ((m (meaning e r))
         (m* (meaning* e* r (length e*))) )
    (lambda (sr k)
      (m sr (lambda (f)
              (if (procedure? f)
                  (m* sr (lambda (v*) (f v* k)))
                  (wrong "Not a function" f) ) )) ) ) )
```

В наших интересах выполнить как можно больше вычислений на этапе предобработки.[1] Например, определить размер необходимой записи активации. Сейчас это сделать элементарно, зная длину списка аргументов. В противоположность этому, гораздо сложнее сказать, *когда* создавать запись. Есть два возможных момента времени:

1) Создавать запись активации до вычисления аргументов. В этом случае вычисленные аргументы кладутся сразу же на нужное место.

2) Создавать запись активации после вычисления всех аргументов. В таком случае, конечно же, аргументы надо будет где-то хранить до того, как они будут размещены в нужной записи активации.

Первый вариант кажется более эффективным, так как потребляет меньше памяти. К сожалению, в Scheme есть `call/cc`, из-за которой этот вариант может работать неверно. Он работает правильно только в Лиспе. Причина лежит в том, что неограниченные продолжения Scheme можно активировать несколько раз. [**см. стр. 109**] Если сначала создать запись активации для вызываемой функции, а потом в процессе вычисления её аргументов захватить продолжение, то все параллельные активации данного продолжения будут пользоваться физически одной и той же записью, — а ведь каждый вызов функции должен иметь свою собственную. Для пояснения проблемы рассмотрим следующий пример. Ожидается, что эта программа вернёт `(2 1)`, но фактически она возвращает `(2 2)` в случае разделяемых записей активаций.[2]

```
(let ((k 'wait)
      (f '()) )
  (set! f (let ((g ((lambda (a) (lambda () a))
                    (call/cc (lambda (nk) (set! k nk) (nk 1))) )))
            (cons g f) ))
  ;; f ≈ (list λ.1)
  (if (null? (cdr f)) (k 2))
  ;; f ≈ (list λ.2 λ.1)
  (list ((car f)) ((cadr f))) )
```

Дело в том, что `call/cc` захватывает аппликацию `((lambda (a) ...) ...)` и её запись активации, если она была создана заранее. Так как продолжение используется дважды, то оба созданных замыкания `(lambda () a)` ссылаются на физически одну и ту же переменную `a`, а значит, активация продолжения `(k 2)` сделает её (в обоих местах) равной двум.

[1] Именно ради этого разработчики языков программирования стараются сделать как можно больше вещей доступными статически.

[2] Мануэль Серрано (Manuel Serrano) обнаружил, что данная программа зависит от порядка вычисления аргументов: форма `cons` должна вычисляться слева направо. Поэтому и используется `let`.

Но ещё не всё потеряно! Записи активаций можно эффективно создавать до вычисления аргументов функции, но тогда `call/cc` придётся самостоятельно делать копии записей активаций при каждой активации продолжения. К сожалению, сейчас она не может этого делать вообще, так как продолжения представляются родными замыканиями языка реализации и извлечь замкнутые в них записи нельзя. Вот ещё один пример того, насколько сильно `call/cc` влияет на реализацию.

Таким образом, функция `meaning*` принимает дополнительный аргумент, указывающий размер необходимой записи активации, которая будет создана после вычисления всех аргументов. [**см. упр. 6.4**] По причинам, которые будут разобраны в разделе 6.1.6 вместе с реализацией функций переменной арности [**см. стр. 236**], записи активаций получают на одно поле больше, чем надо обычным функциям, но это не повлияет на их эффективность, так как данное поле не инициализируется.

```
(define (meaning* e* r size)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size)
      (meaning-no-arguments r size) ) )

(define (meaning-no-arguments r size)
  (let ((size+1 (+ size 1)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

Заметьте, что `size+1` вычисляется во время предобработки — это значение неизменно во время исполнения, поэтому его можно вычислить сразу же. Также обратите внимание на «оседлость» формы, создающей запись активации: она действительно должна вызываться при каждой аппликации.

Будущее расположение аргументов в записи активации легко вычисляется заранее, так что значение каждого аргумента после создания записи тут же укладывается в свою забронированную ячейку.

```
(define (meaning-some-arguments e e* r size)
  (let ((m (meaning e r))
        (m* (meaning* e* r size))
        (index (- size (+ (length e*) 1))) )
    (lambda (sr k)
      (m sr (lambda (v)
              (m* sr (lambda (v*)
                       (set-activation-frame-argument! v* index v)
                       (k v*) )) )) ) ) )
```

Вот теперь мы, наконец, можем определить абстракцию, в точности зная, что происходит с аргументами. Для проверки арности используется длина записи активации, но правильное значение известно заранее, а фактическое

к моменту вызова функции уже вычислено и размещено по известному смещению, так что сейчас нам остаётся только сравнить два готовых числа.

```
(define (meaning-fix-abstraction n* e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ 1 arity))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2)) )
    (lambda (sr k)
      (k (lambda (v* k1)
           (if (= (activation-frame-argument-length v*) arity+1)
               (m+ (sr-extend* sr v*) k1)
               (wrong "Incorrect arity") ) )) ) ) )
```

6.1.4. Классы переменных

Предыдущими определениями поддерживаются только локальные переменные, то есть переменные внутри `lambda`-форм. Но ведь ещё остались глобальные переменные, а среди них — предопределённые и/или неизменяемые вроде `cons` или `car`. Сейчас возможно их поместить разве что в самую дальнюю запись активации, но вы представляете, какой тогда будет скорость доступа к ним. Поэтому мы воспользуемся возможностью статически классифицировать переменные.

Пусть глобальная переменная `g.current` содержит список изменяемых глобальных переменных обрабатываемой программы, а переменная `g.init` — фиксированный список предопределённых неизменяемых переменных: `cons`, `car` и т. д. Функция `compute-kind` определяет класс переменной и возвращает соответствующий дескриптор.

```
(define (compute-kind r n)
  (or (local-variable? r 0 n)
      (global-variable? g.current n)
      (global-variable? g.init n) ) )

(define (global-variable? g n)
  (let ((var (assq n g)))
    (and (pair? var) (cdr var)) ) )
```

`g.current` проверяется перед `g.init`, чтобы оставить возможность при необходимости «переопределять» предопределённые переменные. Естественно, значения этих переменных нельзя и не следует изменять, но стандартом допускается возможность определения одноимённых глобальных переменных при условии, что это не повлияет на работу примитивов. То есть можно определить, например, глобальную переменную `car`, но `map`, которая наверняка использует функцию `car`, должна продолжать работать по-старому. Только явно определяемые в этой же программе функции смогут пользоваться новым значением предопределённой переменной. Одной `compute-kind` достаточно для

использования готовых переопределений, но для создания новых придётся ввести дополнительную специальную форму, скажем, `redefine`. [**см. упр. 6.5**]

Создаются глобальные переменные с помощью функций `g.init-extend!` и `g.current-extend!`.

```
(define (g.current-extend! n)
  (let ((level (length g.current)))
    (set! g.current (cons (cons n `(global . ,level)) g.current))
    level ) )

(define (g.init-extend! n)
  (let ((level (length g.init)))
    (set! g.init (cons (cons n `(predefined . ,level)) g.init))
    level ) )
```

Окружения `g.current` и `g.init` содержат лишь адреса, собственно значения будут располагаться в специально отведённом месте для глобальных переменных — двух векторах, доступных посредством переменных `sg.current` и `sg.init`. Префикс `s` здесь потому, что данные векторы — это всё же кусочек памяти. Теперь определим эти контейнеры[3] и функции-аксессоры к ним. (Конечно, без `predefined-update!`, так как примитивы неизменяемы.)

```
(define sg.current (make-vector 100))
(define sg.init (make-vector 100))

(define (global-fetch i)     (vector-ref sg.current i))
(define (predefined-fetch i) (vector-ref sg.init i))
(define (global-update! i v) (vector-set! sg.current i v))
```

Как обычно, мы определим пару вспомогательных функций `g.current-initialize!` и `g.init-initialize!`, чтобы облегчить наполнение глобальных окружений. Они будут синхронно обновлять глобальные окружения, следя за тем, чтобы определяемая переменная (изначально) находилась только в одном из них.

```
(define (g.current-initialize! name)
  (let ((kind (compute-kind r.init name)))
    (if kind
        (case (car kind)
          ((global)
           (vector-set! sg.current (cdr kind) undefined-value) )
          (else (static-wrong "Wrong redefinition" name)) )
        (let ((index (g.current-extend! name)))
          (vector-set! sg.current index undefined-value) ) ) )
  name )
```

[3] Для простоты количество глобальных переменных пока что ограничено одной сотней. Данное ограничение будет снято в разделе 6.1.9.

```
(define (g.init-initialize! name value)
  (let ((kind (compute-kind r.init name)))
    (if kind
        (case (car kind)
          ((predefined) (vector-set! sg.init (cdr kind) value))
          (else (static-wrong "Wrong redefinition" name)) )
        (let ((index (g.init-extend! name)))
          (vector-set! sg.init index value) ) ) )
  name )
```

Теперь у нас есть полный арсенал функций для реализации предварительной обработки обращений к переменным и присваиваний. Оба обработчика имеют весьма схожую структуру: выяснить класс переменной и подать правильный аксессор. Заметьте, что функции `deep-fetch` и `deep-update!` не используются, если переменная находится на верхушке стека записей активаций — в этом случае значения получаются и изменяются напрямую. Также здесь применяется одна искусная, но спорная уловка: предопределённые переменные считаются чем-то вроде цитат: обращения к ним сразу же заменяются соответствующими значениями, а не выполняют вызовы аксессоров.

```
(define (meaning-reference n r)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (lambda (sr k)
                   (k (activation-frame-argument sr j)) )
                 (lambda (sr k)
                   (k (deep-fetch sr i j)) ) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (if (eq? (global-fetch i) undefined-value)
                 (lambda (sr k)
                   (let ((v (global-fetch i)))
                     (if (eq? v undefined-value)
                         (wrong "Uninitialized variable" n)
                         (k v) ) ) )
                 (lambda (sr k)
                   (k (global-fetch i)) ) ) ) )
          ((predefined)
           (let* ((i (cdr kind))
                  (value (predefined-fetch i)) )
             (lambda (sr k) (k value)) ) ) )
        (static-wrong "No such variable" n) ) ) )
```

Присваивание аналогично:

```
(define (meaning-assignment n e r)
  (let ((m (meaning e r))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (lambda (sr k)
                   (m sr (lambda (v)
                           (k (set-activation-frame-argument!
                               sr j v )) )) )
                 (lambda (sr k)
                   (m sr (lambda (v)
                           (k (deep-update! sr i j v)) )) ) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (lambda (sr k)
               (m sr (lambda (v)
                       (k (global-update! i v)) )) ) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

Статические ошибки

Одна из целей предварительной обработки программ — это заранее отыскать все глупые ошибки вроде обращений к явно несуществующим переменным или попыток изменения значений предопределённых переменных. Ведь без обработки они остаются незамеченными, пока ошибочный код не вызывается. О таких ошибках сигнализирует форма `static-wrong`, тогда как `wrong` служит для ошибок времени исполнения, которые нельзя предвидеть заранее. Идея статических ошибок (ошибок времени компиляции) очень полезна, но в то же время проводит чёткую границу между либеральными языками вроде Лиспа и большей частью остальных. К примеру, любая корректная программа на ML гарантированно лишена ошибок типизации. В Лиспе же всё наоборот: только гарантированно ошибочные программы считаются некорректными. Рассмотрим следующую функцию:

```
(define (statically-strange n)
  (if (integer? n)
      (if (= n 0) 1 (* n (statically-strange (- n 1))))
      (cons) ) )
```

Несмотря на то, что статически (в общем случае) эта функция ошибочна, она может быть весьма полезной, если ей передавать целые числа (особенно положительные!). Разрешать подобные определения или нет — это вопрос духа языка, это компромисс между желаемой безопасностью и отдаваемой за неё свободой. Очевидно, что о возможных ошибках лучше быть извещённым как можно раньше — ML возводит эту максиму в абсолют, — однако, как показывает Лисп, вполне можно жить пусть и с постоянной опасностью ошибиться, но получая взамен неимоверную гибкость языка.

Поэтому функция `static-wrong` должна, конечно же, сообщать об ошибке, но при этом обязана возвращать нормальный результат предобработки. Конечно, вся его работа сводится к сообщению о том, что мы таки наступили на грабли. Предупреждение можно вывести заранее, но *возникнуть* ошибка может только во время исполнения программы. Мы явно описываем желаемое поведение в определении `static-wrong`:

```
(define (static-wrong message . culprits)
  (display `(*static-error*
             ,message . ,culprits) )
  (newline)
  (lambda (sr k)
    (apply wrong message culprits) ) )
```

Как говорится, и рыбку съели, и косточкой не подавились.

Кстати, вспомните, что при доступе к глобальным переменным необходимо проверять их инициализированность, чего не требуется делать для локальных и предопределённых. Следовательно, есть некоторая дополнительная стоимость обращения к глобальным изменяемым переменным. В случае компиляции на лету (например, при выполнении чего-то вроде `(display (compile-and-run (read)))`) мы могли бы немного оптимизировать обращения к инициализированным глобальным переменным. В действительности, у нас даже сейчас есть подобная оптимизация[4] в `meaning-reference`. [**см. упр. 7.6**]

6.1.5. Запускаем интерпретатор

Интерпретатор читает выражение, обрабатывает его, затем исполняет и повторяет всё сначала. Таким образом, мы получаем компилирующий REPL.

```
(define  r.init '())
(define sr.init '())

(define (chapter6.1-interpreter)
  (define (compile e) (meaning e r.init))
  (define (run c) (c sr.init display))
```

[4] Если бы у нас была возможность изменять скомпилированный код на лету, то можно было бы пойти и дальше: после инициализации переменной просто выкинуть код проверок на инициализированность изо всех обращений к ней. Например, так делает Bigloo [Ser94].

```
(define (toplevel)
  (run (compile (read)))
  (toplevel) )
(toplevel) )
```

Конечно, перед его запуском стоит всё же хоть немного наполнить базовое окружение. Как обычно, предполагается, что у нас есть готовые макросы, которые выполнят всю чёрную рутинную работу, позволяя сфокусироваться на главном. Вот несколько подобных определений; среди них, конечно же, и наша ненаглядная `call/cc`.

```
(definitial t #t)
(definitial f #f)
(definitial nil '())

(defprimitive cons cons 2)
(defprimitive car car 1)

(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (lambda (v* k)
      (if (= arity+1 (activation-frame-argument-length v*))
          ((activation-frame-argument v* 0)
           (let ((frame (allocate-activation-frame (+ 1 1))))
             (set-activation-frame-argument!
              frame 0
              (lambda (values kk)
                (if (= (activation-frame-argument-length values)
                       arity+1 )
                    (k (activation-frame-argument values 0))
                    (wrong "Incorrect arity" 'continuation) ) ) )
             frame )
           k )
          (wrong "Incorrect arity" 'call/cc) ) ) ) )
```

6.1.6. Функции с переменной арностью

В нашем интерпретаторе всё ещё не поддерживаются функций с переменной арностью. Как всегда, реализация подобных функций представляет некоторые трудности. Наши записи активаций хранят значения аргументов, но они же фактически являются и вместилищем привязок. Функции переменной арности нарушают правило «один аргумент — одна привязка». Например, функция, чьи аргументы описываются списком `(a b . c)`, может принять два, три, четыре или десять аргументов, но привязок во всех случаях будет ровно три. По этой причине в записи активации такой функции должно быть

как минимум три поля. Аналогично для аппликации (*f a b*) тоже надо создавать запись активации с тремя полями, чтобы ей могла воспользоваться любая функция, которая может принять два аргумента: то есть функции со списками аргументов вида `(x y)`, `(x y . z)`, `(x . y)`, или даже просто `x`.

Соответственно, функции с переменной арностью просто сбрасывают в список все «лишние» аргументы, которые они найдут в записях активаций. Такую обработку будет выполнять специально обученная функция `listify!`. Она несложная, но, требует определённой эквилибристики с перераспределением переменных. Её параметр `arity` указывает минимальное количество принимаемых функцией аргументов.

```
(define (meaning-dotted-abstraction n* n e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ 1 arity))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence e+ r2)) )
    (lambda (sr k)
      (k (lambda (v* k1)
           (if (>= (activation-frame-argument-length v*) arity+1)
               (begin (listify! v* arity)
                      (m+ (sr-extend* sr v*) k1) )
               (wrong "Incorrect arity") ) )) ) ) )

(define (listify! v* arity)
  (let loop ((index (- (activation-frame-argument-length v*) 1))
             (result '()) )
    (if (= arity index)
        (set-activation-frame-argument! v* arity result)
        (loop (- index 1)
              (cons (activation-frame-argument v* (- index 1))
                    result ) ) ) ) )
```

Наконец, всё готово для предобработки обоих видов `lambda`-форм. Выбор необходимого обработчика легко выполняется статически:

```
(define (meaning-abstraction nn* e+ r)
  (let parse ((n* nn*)
              (regular '()) )
    (cond
     ((pair? n*) (parse (cdr n*) (cons (car n*) regular)))
     ((null? n*) (meaning-fix-abstraction nn* e+ r))
     (else (meaning-dotted-abstraction (reverse regular) n* e+ r)) ) ) )
```

6.1.7. Приводимые выражения

Давайте реализуем в нашем интерпретаторе известную оптимизацию, касающуюся приводимых форм (редексов): аппликаций, первым термом которых

является `lambda`-форма. В этом случае не требуется создавать замыкание, достаточно только ввести на время новые локальные переменные как в блоках Алгола. В конце концов, `((lambda (...) ...) ...)` — это не что иное, как `let`-форма: всего лишь блок кода с локальными переменными. И снова, для функций фиксированной арности всё легко и просто, чего не скажешь об арности переменной.[5] Кстати, в этом случае неправильное количество аргументов является статически обнаруживаемой ошибкой.

```
(define (meaning-closed-application e ee* r)
  (let ((nn* (cadr e)))
    (let parse ((n* nn*)
                (e* ee*)
                (regular '()) )
      (cond ((pair? n*)
             (if (pair? e*)
                 (parse (cdr n*) (cdr e*) (cons (car n*) regular))
                 (static-wrong "Too few arguments" e ee*) ) )
            ((null? n*)
             (if (null? e*)
                 (meaning-fix-closed-application
                  nn* (cddr e) ee* r )
                 (static-wrong "Too many arguments" e ee*) ) )
            (else
             (meaning-dotted-closed-application
              (reverse regular) n* (cddr e) ee* r ) ) ) ) ) ) )

(define (meaning-fix-closed-application n* body e* r)
  (let* ((m* (meaning* e* r (length e*)))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence body r2)) )
    (lambda (sr k)
      (m* sr (lambda (v*)
               (m+ (sr-extend* sr v*) k)) ) ) ) )
```

Для функций с переменной арностью можно не вызывать `listify!`, так как минимальная арность и фактическое число аргументов известны статически. Специальный вариант `meaning*`, названный `meaning-dotted*`, отвечает за обработку данного случая. Он работает подобно `meaning*` для обязательных аргументов, а дополнительные на лету собирает в список, искусно расставляя `cons`. Конечно, это не дёшево, но за удобства надо платить. Также придётся вручную инициализировать последний элемент записи активации значением `()`. В общем, он выполняет вот такое преобразование:

```
((lambda (a b . c) ...) ⇝ ((lambda (a b c) ...)
 α β γ δ ...)               α β (cons γ (cons δ ...)) )
```

[5] Это уже который раз мы такое говорим? Надеюсь, теперь вам понятно, почему во многих языках подобные функции не реализованы вообще, несмотря на всю их полезность.

с помощью вот таких функций:

```
(define (meaning-dotted-closed-application n* n body e* r)
  (let* ((m* (meaning-dotted* e* r (length e*) (length n*)))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence body r2)) )
    (lambda (sr k)
      (m* sr (lambda (v*)
               (m+ (sr-extend* sr v*) k)) ) ) ) )

(define (meaning-dotted* e* r size arity)
  (if (pair? e*)
      (meaning-some-dotted-arguments (car e*) (cdr e*) r size arity)
      (meaning-no-dotted-argument r size arity) ) )

(define (meaning-some-dotted-arguments e e* r size arity)
  (let ((m (meaning e r))
        (m* (meaning-dotted* e* r size arity))
        (index (- size (+ (length e*) 1))) )
    (if (< index arity)
        (lambda (sr k)
          (m sr (lambda (v)
                  (m* sr (lambda (v*)
                           (set-activation-frame-argument! v* index v)
                           (k v*) )) )) )
        (lambda (sr k)
          (m sr (lambda (v)
                  (m* sr (lambda (v*)
                           (set-activation-frame-argument!
                            v* arity
                            (cons v (activation-frame-argument
                                     v* arity )) )
                           (k v*) )) ) ) ) ) )

(define (meaning-no-dotted-argument r size arity)
  (let ((arity+1 (+ 1 arity)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        (k v*) ) ) ) )
```

6.1.8. Интеграция примитивов

Мы можем серьёзно ускорить интерпретацию, особым образом реализовав предобработку вызовов примитивных предопределённых функций. Сейчас аппликация вроде `(car` α`)` вызывает ужасно расточительную кучу действий:

1) Разыменовать глобальную переменную `car`.

2) Вычислить α.

3) Создать запись активации с двумя полями.

4) Положить в первое поле записи значение α.

5) Убедиться, что значением `car` действительно является функция.

6) Убедиться, что эта функция действительно может принять один аргумент.

7) Применить значение `car` к значению α. Вдобавок функция `car` ещё проверяет, что её аргумент действительно является точечной парой.

Некоторые из этих шагов не нужны в статически типизированных языках, именно поэтому такие языки очень быстрые. Но при определённых усилиях можно избавиться от некоторых проверок и в Лиспе, выполняя их во время предобработки. Так как `car` это глобальная переменная, чьё значение нельзя изменить, то мы избавляемся от шага 5 (убедиться, что это функция) и шага 6 (проверить арность). Можно сэкономить ещё больше, если не создавать запись активации (шаги 3 и 4), а сразу вставить код вызываемого примитива в программу (шаг 1).

Подобный способ использования примитивов — прямое исполнение кода в обход протокола вызова функций — называется *инлайнинг*. В этом случае остаётся выполнить только шаги 2 и 7. Вдобавок, хороший компилятор будет выполнять седьмой шаг по-умному, избегая дублирования проверок типов. Например, в выражении `(if (pair? e) (car e) ...)` бессмысленно проверять тип `e` внутри `car`, так как он очевидно и гарантированно правильный. Один из способов реализации такого поведения следующий: считать `(car` x`)` макросом, раскрывающимся в `(let ((`v x`)) (if (pair?` v`) (unsafe-car` v`) (error ...)))`, где `unsafe-car`[6] работает как `car`, если её аргумент это точечная пара, и имеет неопределённое поведение в противном случае. Теперь остаётся только перенести все проверки типов как можно выше, оставив таким образом лишь неизбежные и избавившись от очевидных повторений. С миграцией проверок типов проблем нет, так как сами по себе они не вызывают ошибок и бесконечных циклов.

При настоящей компиляции мы не имеем непосредственного доступа к значениям предопределённых переменных: они обычно лежат где-то в библиотеке времени исполнения. Однако, для предварительной обработки программ

[6] Большинство реализаций предоставляют подобные небезопасные примитивы, оказывающиеся весьма полезными для различных преобразований программ. Естественно, эти преобразования должны гарантировать использование `unsafe`-примитивов только в безопасном контексте. Компилятор часто способен уследить за типами значений, хранимых в переменных, поэтому данные примитивы очень важны для эффективной компиляции.

эти значения не особо-то и нужны, достаточно будет знать, функция ли это и сколько аргументов она может принять. Поэтому мы добавим новое окружение: окружение дескрипторов глобальных предопределённых переменных `desc.init`. В нём будут храниться описания предопределённых примитивных функций, представляемые списками. Первый элемент такого списка — это символ `function`, второй — «адрес» примитива (то, что вызывается), а за ним следует описание арности. Макрос `defprimitive` используется для определения примитивов. Здесь приведён только один из его подмакросов: `defprimitive3`, определения остальных аналогичны.

```
(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name value 0) (defprimitive0 name value))
    ((defprimitive name value 1) (defprimitive1 name value))
    ((defprimitive name value 2) (defprimitive2 name value))
    ((defprimitive name value 3) (defprimitive3 name value)) ) )

(define-syntax defprimitive3
  (syntax-rules ()
    ((defprimitive3 name value)
     (definitial name
       (letrec ((arity+1 (+ 3 1))
                (behavior
                 (lambda (v* k)
                   (if (= (activation-frame-argument-length v*)
                          arity+1 )
                       (k (value (activation-frame-argument v* 0)
                                 (activation-frame-argument v* 1)
                                 (activation-frame-argument v* 2) ))
                       (wrong "Incorrect arity" 'name) ) ) ) )
         (description-extend!
          'name `(function ,value a b c))
         behavior ) ) ) ) )
```

Управление новым окружением возлагается на следующие функции:

```
(define desc.init '())

(define (description-extend! name description)
  (set! desc.init (cons (cons name description) desc.init))
  name )

(define (get-description name)
  (let ((p (assq name desc.init)))
    (and (pair? p) (cdr p)) ) )
```

Вот теперь можно действительно полностью определить обработку аппликаций. Она сводится к определению класса вызываемой функции и передаче

аппликации соответствующему специфичному обработчику. Заметьте, что согласованность арности для примитивов тоже проверяется статически.

```
(define (meaning-application e e* r)
  (cond
   ((and (symbol? e)
         (let ((kind (compute-kind r e)))
           (and (pair? kind)
                (eq? 'predefined (car kind))
                (let ((desc (get-description e)))
                  (and desc
                       (eq? 'function (car desc))
                       (if (= (length (cddr desc)) (length e*))
                           (meaning-primitive-application e e* r)
                           (static-wrong "Incorrect arity" e) ) ) ) ) ) ) ))
   ((and (pair? e)
         (eq? 'lambda (car e)) )
    (meaning-closed-application e e* r) )
   (else (meaning-regular-application e e* r)) ) )
```

Аппликации примитивов обрабатываются следующим образом:

```
(define (meaning-primitive-application e e* r)
  (let* ((desc (get-description e)) ; desc = (function адрес . аргументы)
         (address (cadr desc))
         (size (length e*)) )
    (case size
     ((0) (lambda (sr k) (k (address))))
     ((1) (let ((m1 (meaning (car e*) r)))
            (lambda (sr k)
              (m1 sr (lambda (v)
                       (k (address v)) )) ) ))
     ((2) (let ((m1 (meaning (car e*) r))
                (m2 (meaning (cadr e*) r)) )
            (lambda (sr k)
              (m1 sr (lambda (v1)
                (m2 sr (lambda (v2)
                         (k (address v1 v2)) )) ) ) )))
     ((3) (let ((m1 (meaning (car e*) r))
                (m2 (meaning (cadr e*) r))
                (m3 (meaning (caddr e*) r)) )
            (lambda (sr k)
              (m1 sr (lambda (v1)
                (m2 sr (lambda (v2)
                  (m3 sr (lambda (v3)
                           (k (address v1 v2 v3)) )) )) ) ) ))
     (else (meaning-regular-application e e* r)) ) ) )
```

Данный вариант поддерживает только примитивы фиксированной арности. Большая часть примитивов, обладающих переменной арностью, вроде `append`, `for-each`, `list`, `map`, `*`, `+` и многие другие (но не `apply`), могут считаться чем-то вроде макросов: например, `(append` π_1 π_2 π_3`)` раскрывается в `(append` π_1 `(append` π_2 π_3`))`. Такая трансформация позволяет передавать произвольное число аргументов, но спасает не всегда. Например, в случае `(apply append` π`)` примитивная функция `append` уже вынуждена поддерживать произвольное число аргументов самостоятельно.

Мы заменяем инлайн-версиями только аппликации с тремя и менее аргументами. Причина такого решения банальна: в Scheme просто нет примитивных функций с фиксированной арностью, принимающих больше трёх аргументов!

6.1.9. Вариации на тему окружений

В данный момент время доступа к глубоко расположенным локальным переменным (к тем, которые не находятся в первой записи активации) линейно зависит от их расположения, так как мы вынуждены каждый раз заново проходить по списку записей активаций. Существует простой способ получать доступ ко всем переменным за постоянное время: поместить в каждую запись активации *дисплей*, хранящий прямые ссылки на все записи, расположенные глубже (см. рисунок 6.3). В таком случае достаточно будет одной косвенной адресации (через ячейку дисплея до записи) и одного сдвига (до нужного значения в индексированном поле записи). Ценой за повышение скорости обращений является увеличение потребления памяти, а также замедление создания новых записей активаций. Конечно, в дисплее можно хранить ссылки не на все записи, а только на те, которые действительно используются, но это требует соответствующего анализа всех обращений к переменным. К сожалению, мы не можем прямо воспользоваться этим приёмом в нашем интерпретаторе, так как сейчас размер записи активации фиксируется до вызова функции, а при использовании дисплеев он будет известен только во время исполнения. Необходимо или хранить дисплеи отдельно от записей, или ограничить максимально возможную высоту стека (что не особо в духе Лиспа).

Плоские окружения

Другой вариант решения проблемы скорости доступа: избавиться от полноценного стека окружений. Для каждого создаваемого замыкания можно не просто слепо сохранять внутри всё состояние окружения `sr`, а формировать для него специальную запись активации, в которой собираются все его свободные переменные. В этом случае, конечно, чуть возрастает стоимость создания замыканий, но все свободные переменные будут доступны за постоянное время, так как в окружениях будет всего по две записи активации: одна с собственными локальными переменными активного замыкания, а другая —

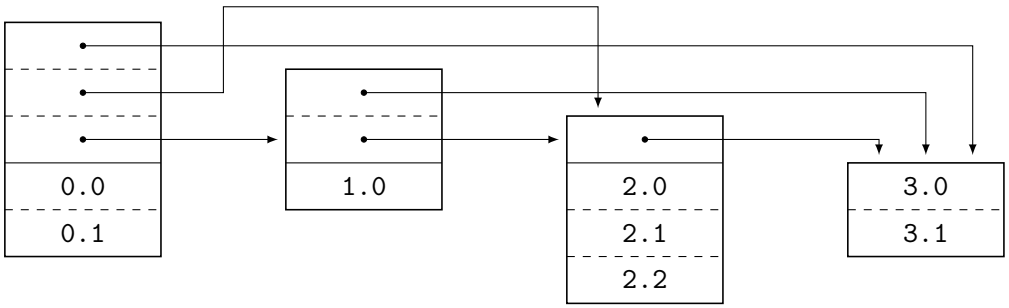


Рис. 6.3. Структура дисплеев.

со свободными. Конечно, на случай, если потребуется разделять какую-либо переменную между несколькими замыканиями, имеет смысл реализовать коробки. [**см. стр. 146**]

В идеале, естественно, стоит использовать все возможные варианты, применяя каждый из них в тех случаях, для которых он работает лучше остальных. Чтобы осуществлять такое адаптивное поведение, требуется более тщательный анализ программ. Это работа для настоящего компилятора, а не для простого интерпретатора с небольшой предобработкой вроде нашего.

Определение глобальных переменных

Наш интерпретатор считает, что переменная глобальная, если она находится в списке глобальных изменяемых или же предопределённых переменных. Следовательно, перед использованием переменную надо поместить в один из этих списков, иными словами: объявить. Внутри интерпретатора это выполняется с помощью макроса `defvariable`:

```
(define-syntax defvariable
  (syntax-rules ()
    ((defvariable name)
     (g.current-initialize! 'name) ) ) )
```

Многие языки имеют что-то подобное для явного объявления переменных, но в большинстве диалектов Лиспа принят другой подход. Здесь правильным считается автоматическое создание всех используемых глобальных переменных. Один из наиболее простых способов этого добиться: поместить механизм создания глобальных переменных прямиком в функцию `compute-kind`:

```
(define (compute-kind r n)
  (or (local-variable? r 0 n)
      (global-variable? g.current n)
      (global-variable? g.init n)
      (adjoin-global-variable! n) ) )
```

```
(define (adjoin-global-variable! name)
  (let ((index (g.current-extend! name)))
    (vector-set! sg.current index undefined-value)
    (cdr (car g.current)) ) )
```

Но так делать не совсем хорошо, потому что тогда предобработка берёт на себя слишком много. Получается, что глобальные переменные создаются во время неё, а не во время исполнения программы. Приемлемо ли такое предрождение? Давайте ещё раз разберём, что именно происходит при упоминании новой переменной.

1) Её имя добавляется в окружение глобальных изменяемых переменных (известное как `g.current`), а также с переменной связывается уникальный номер.

2) Для хранения значения переменной выделяется ячейка памяти с адресом, соответствующим её номеру.

Держа это в уме, рассмотрим следующую программу:

```
(begin (define foo (lambda () (bar)))
       (define bar (lambda () (if #t 33 hux))) )
```

Так как в процессе её предобработки были встречены неизвестные переменные с именами `foo`, `bar` и `hux`, то они все получают свои номера, ячейки памяти и добавляются в окружение. Но как их существование соотносится с семантикой обрабатываемой программы? Почему определение `foo` игнорирует явно заданный порядок вычисления форм? Зависит ли существование `hux` от истинности условия?

Как бы то ни было, во время исполнения программы значения всех используемых глобальных переменных должны находиться в глобальном окружении. Идеальный момент для их добавления туда — это непосредственно перед началом исполнения программы. Вернее, это единственно возможный момент. Лишь тогда размер глобального окружения можно окончательно подогнать под требуемое количество глобальных переменных, а также выдать им начальные значения. Это выполняет функция `standalone-producer`, которая обрабатывает программу, возвращая обычное замыкание `(lambda (sr k) ...)`, первым действием которого является создание личного глобального окружения необходимых размеров.

```
(define (standalone-producer e)
  (set! g.current (original.g.current))
  (let* ((m (meaning e r.init))
         (size (length g.current)) )
    (lambda (sr k)
      (set! sg.current (make-vector size undefined-value))
      (m sr k) ) ) )
```

В глобальном окружении могут быть и предопределённые изменяемые переменные, служащие механизмом обмена информацией между программой и внешней системой. Они могут содержать разнообразные настройки вроде основания счисления для вывода чисел. Предполагается, что такие переменные содержатся в окружении, возвращаемом `original.g.current`.

Итого, при такой реализации для создания переменной достаточно лишь упомянуть её имя. Можно было бы поступить наоборот, как сделано в Scheme, где глобальные переменные создаются исключительно формой `define`. В этом случае предобработка примера с `foo` должна выдать ошибку, так как `hux` — это неопределённая переменная с точки зрения предобработки, и не важно, что фактически к ней никогда даже не обратятся. Переменная `bar` проблем не вызывает, так как её определение в программе присутствует (пусть и чуть дальше).

Для пояснения принципа работы `define` мы добавим новую специальную форму в `meaning`:

```
... ((define) (meaning-define (cadr e) (caddr e) r)) ...
```

При обработке форм `define` обнаруживаются все повторные определения переменных. Хоть переопределение переменных не является ошибкой, но лучше предупредить пользователя. Так как все глобальные переменные уже будут существовать перед выполнением программы, то все определения можно просто заменять присваиваниями.

```
(define (meaning-define n e r)
  (let ((b (memq n *defined*)))
    (if (pair? b)
        (static-wrong "Already defined variable" n)
        (set! *defined* (cons n *defined*)) ) )
  (meaning-assignment n e r) )
```

Также на предобработку возлагается дополнительная задача: проверить, все ли упоминаемые глобальные переменные были определены и получили значения. Естественно, в список определённых входят также и предопределённые переменные.

```
(define (standalone-producer e)
  (set! g.current (original.g.current))
  (let ((originally-defined (append (map car g.current)
                                    (map car g.init) )))
    (set! *defined* originally-defined)
    (let* ((m (meaning e r.init))
           (size (length g.current))
           (anormals (set-difference (map car g.current) *defined*)) )
      (if (null? anormals)
          (lambda (sr k)
            (set! sg.current (make-vector size undefined-value))
            (m sr k) )
          (static-wrong "Undefined global variables" anormals) ) ) ) )
```

```
(define (set-difference set1 set2)
  (if (pair? set1)
      (if (memq (car set1) set2)
          (set-difference (cdr set1) set2)
          (cons (car set1) (set-difference (cdr set1) set2)) )
      '() ) )
```

Подводя итог, необходимо чётко разделять предобработку и исполнение программ. Во время исполнения программы определения эквивалентны присваиваниям. Но во время предобработки они требуют больших усилий, проверок и так далее. Обычные специальные формы (не `define`) действуют исключительно во время исполнения программ. Их семантика динамическая. Определение переменных — это статический процесс (благодаря предобработке), потому как все глобальные переменные должны быть определены с помощью специальной формы `define`, чьё наличие можно проверить статически, а отсутствие считается ошибкой.

Кроме того, как мы упоминали ранее в разговоре о `letrec` [**см. стр. 83**], важно понимать, что создание и инициализация переменных — это не одно и то же. Переменная может существовать, но не иметь при этом значения. Вот вам пример такой переменной:

```
(begin (define foo bar)
       (define bar (lambda () 33)) )
```

В случае компилирующего интерпретатора (нечто вроде инкрементального компилятора, сразу же вычисляющего скомпилированные им выражения) большая часть рассмотренных проблем пропадает сама собой благодаря тому, что программа выполняется сразу же после её обработки и, что самое главное, с тем состоянием памяти, которое было на конец обработки. Поэтому здесь первый вариант `adjoin-global-variable!`, сразу же добавляющий переменные в память, в принципе подходит и работает правильно. То же касается и улучшения в `meaning-reference`, удаляющего проверки на инициализированность для уже инициализированных переменных.

6.1.10. Выводы: интерпретатор с миграцией вычислений

Довольно сложно точно измерить достигнутое увеличение эффективности, но оно составляет порядка 10—50 раз. Предварительная обработка выполняется достаточно быстро, но возможности для оптимизации ещё остаются (самое очевидное: представлять окружения не списками, а хеш-таблицами). Кроме того, немаловажен и новый подход с разделением вычислений на статическую и динамическую части. Хороший компилятор должен как можно больше всего выполнять статически, чтобы во время исполнения программа делала как можно меньше ненужных действий. Естественно, непрестанно ведутся иссле-

дования, нацеленные на поиск новых статических свойств программ, которые могли бы улучшить качество предварительной обработки. Вот лишь некоторые из них: абстрактная интерпретация в работе [CC77], частичные вычисления из [JGS93], анализ потока исполнения в [Shi91]. Ещё одним аспектом, влияющим на эффективность, является выбор используемых структур данных, как можно было увидеть из нашей дискуссии о представлении записей активаций.

Предварительная обработка позволяет отловить некоторые ошибки заранее, до того, как они возникнут при исполнении программы. Это первый шаг на пути к более безопасному и эффективному программированию: меньше возможных ошибок — меньше явных проверок. Рассмотренная здесь обработка довольно примитивна, как минимум её можно научить проверять типы, как это сделано в ML.

Полученный интерпретатор очень напоминает рассмотренный в третьей главе, разве что замыкания в нашем языке реализуются напрямую с помощью родных замыканий Scheme. Несмотря на то, что объекты и замыкания во многом схожи, замыкания всё же недостаточно выразительны для данного случая: у них только один метод — вызов; их внутренняя структура покрыта мраком; их поведение невозможно уточнить или изменить, не говоря уже о более сложных вещах вроде рефлексии.

6.2. Отказ от окружений

Каждое выражение вычисляется в так или иначе уникальном окружении, у нас ему соответствует значение параметра `sr`. Так как окружение изменяется исключительно при вызовах функций — они восстанавливают окружение своего создания, после чего расширяют его локальными переменными, — то, вероятно, имеет смысл ввести явное понятие текущего состояния окружения, храня его в глобальной переменной (или регистре) `*env*`. Это позволило бы избежать передачи окружения через аргумент всем замыканиям, получающимся в результате предобработки. Иными словами, в таком случае предварительная обработка будет возвращать замыкания всего с одним параметром: `(lambda (k) ...)`.

На первый взгляд, для выполнения подобного преобразования, заменяющего локальные переменные функций `sr` одной глобальной переменной `*env*`, достаточно внести лишь косметические изменения:

1) Отныне локальные переменные следует искать в `*env*`, а не в `sr`.

2) Замыкания сохраняют расширенные окружения своего определения и при своём вызове просто устанавливают в `*env*` нужное им значение.

Однако мы пропустили одну маленькую, но существенную деталь: время от времени необходимо восстанавливать старое окружение, а именно, когда

происходит возврат к вычислениям, прерванным вызовом функции. Решение в лоб: функция при вызове сохраняет текущее значение `*env*`, а при возврате восстанавливает его. Но вот здесь возникает проблема с тем, что Scheme требует оптимизации хвостовых вызовов, то есть иногда `*env*` восстанавливать *не надо*, потому что хвостовые вызовы должны выполняться с неизменным продолжением.

Эта проблема напоминает проблему выбора протокола вызова функций на самом низком, машинном уровне. При вызове необходимо сохранять и восстанавливать регистры процессора, но кто это должен делать?

- Вызываемая подпрограмма точно знает, какие регистры ей понадобятся, так что может сохранять только их. Сложность тут в том, что восстановление регистров необходимо выполнять непосредственно перед возвратом управления из подпрограммы. Другими словам, это становится частью её продолжения.

- Вызывающий подпрограмму в свою очередь знает, какие регистры понадобятся ему, поэтому может сохранить их самостоятельно. В этом случае подпрограмма должна лишь выполниться и положить результат туда, где его ждут. Никаких проблем с продолжениями, и если вызывающему не важно состояние регистров, то можно вообще ничего не делать. Правда, подпрограмме может быть нужна лишь пара регистров, но вызывающему об этом она никак не скажет.

В [SS80] предлагается следующее решение: вызывающий помечает регистры метками «сохранить при использовании» и «можно свободно изменять», а вызываемый может сохранять только то, что действительно надо сохранять, помечая регистры как «восстановить при возврате» и «значение не менялось». Тогда регистры становятся фактически стеками значений, где верхнее значение является текущим, а под ним лежат дожидающиеся восстановления старые значения. Во время возврата управления специальная машинная инструкция обновляет значения регистров в соответствии с расставленными метками.

Очень важно, чтобы хвостовые вызовы проводились с неизменным продолжением (то есть не вызывали роста стека), это позволяет эффективно реализовывать итерацию через рекурсию, не ограничивая возможную длину цикла размером стека. Поэтому мы будем использовать вторую стратегию: ответственным за сохранение (или несохранение) окружения является тот, кто вызывает функцию.

К счастью, найти все вызовы из хвостовых позиций можно статически, так что функция `meaning` получает дополнительный аргумент: булево значение `tail?`, показывающее, расположено ли вычисляемое выражение в хвостовой позиции или нет. Если да, то можно не восстанавливать окружение после завершения вычислений. (Естественно, в таком случае его и сохранять не обязательно.)

Так где же у выражений хвост? Для ответа на этот вопрос следует обратиться к денотациям: всё, что вычисляется с продолжением, отличным от продолжения внешней формы, находится не в хвостовой позиции. В присваивании `(set!` x π`)` форма π имеет продолжение, отличное от такового у формы `set!`, так как ещё остаётся записать её значение в переменную x. Следовательно, это не хвостовой вызов. Аналогично, форма π_0 в ветвлении `(if` π_0 π_1 π_2`)` тоже не является последним вычислением. В последовательности `(begin` π_0 `...` π_{n-1} π_n`)` формы $\pi_0, \ldots, \pi_{n-1}$ требуется вычислять, передавая изменяющееся окружение от одной к следующей, так что это не хвостовые вызовы. В аппликации `(`π_0 `...` π_n`)` ни одна форма не находится в хвостовой позиции, так как ещё остаётся собственно вызов функции. Однако вычисление тела функции, равно как и выполнение всей программы, находятся как раз в хвостовой позиции, потому что ни тело функции само по себе, ни вся программа не должны восстанавливать окружение после своего завершения.

Вот ядро нашего интерпретатора: новая функция `meaning` и главный цикл `toplevel`.

```
(define (meaning e r tail?)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r tail?)
                      (meaning-quotation e r tail?) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r tail?))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r tail?))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)
                                       r tail? ))
        ((begin)  (meaning-sequence (cdr e) r tail?))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r tail?))
        (else     (meaning-application (car e) (cdr e) r tail?)) ) ) )

(define *env* sr.init)
(define (chapter6.2-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    ((meaning (read) r.init #t) display)
    (toplevel) )
  (toplevel) )
```

6.2.1. Обращения к переменным

Для определения значения переменной теперь используется регистр `*env*`. Мы не будем приводить определение присваивания, так как изменения в нём вполне очевидны.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
```

```
(if kind
    (case (car kind)
      ((local)
       (let ((i (cadr kind))
             (j (cddr kind)) )
         (if (= i 0)
             (lambda (k)
               (k (activation-frame-argument *env* j)) )
             (lambda (k)
               (k (deep-fetch *env* i j)) ) ) ) )
      ((global)
       (let ((i (cdr kind)))
         (if (eq? (global-fetch i) undefined-value)
             (lambda (k)
               (let ((v (global-fetch i)))
                 (if (eq? v undefined-value)
                     (wrong "Uninitialized variable" n)
                     (k v) ) ) )
             (lambda (k)
               (k (global-fetch i)) ) ) ) )
      ((predefined)
       (let* ((i (cdr kind))
              (value (predefined-fetch i)) )
         (lambda (k)
           (k value) ) ) ) )
    (static-wrong "No such variable" n) ) ) )
```

6.2.2. Ветвление

Цитирование мы тоже пропустим, так как ему без разницы, где находиться. Что касается условного оператора, то вы помните, что условие должно вычисляться с восстановлением окружения.

```
(define (meaning-alternative e1 e2 e3 r tail?)
  (let ((m1 (meaning e1 r #f))        ; хвост чуть дальше!
        (m2 (meaning e2 r tail?))
        (m3 (meaning e3 r tail?)) )
    (lambda (k)
      (m1 (lambda (v)
            ((if v m2 m3) k)) ) ) ) )
```

6.2.3. Последовательность

Последнее выражение в последовательности должно сохранять окружение, если вся последовательность должна это делать. Остальные обязаны его

сохранять и восстанавливать, но только если они сами его изменяют — а изменять окружение могут только аппликации абстракций. Например, в последовательности `(begin a (car x) ...)` окружение можно не сохранять, так как ни `a`, ни `(car x)` не способны его изменить.

```
(define (meaning-sequence e+ r tail?)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r tail?)
          (meaning*-single-sequence (car e+) r tail?) )
      (static-wrong "Illegal syntax: (begin)") ) )

(define (meaning*-single-sequence e r tail?)
  (meaning e r tail?) )

(define (meaning*-multiple-sequence e e+ r tail?)
  (let ((m1 (meaning e r #f))
        (m2 (meaning-sequence e+ r tail?)) )
    (lambda (k)
      (m1 (lambda (v)
            (m+ k) )) ) ) )
```

6.2.4. Абстракция

Замыкания захватывают окружение своего создания. Они же обязаны установить захваченное окружение текущим, расширить его локальными переменными и передать управление телу функции. Тело функции не должно восстанавливать окружение. Здесь мы рассмотрим только функции с фиксированной арностью, обработка точечных аргументов не влияет на концепцию хвостовых вызовов.

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (arity+1 (+ 1 arity))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (lambda (k)
      (let ((sr *env*))
        (k (lambda (v* k1)
             (if (= (activation-frame-argument-length v*) arity+1)
                 (begin (set! *env* (sr-extend* sr v*))
                        (m+ k1) )
                 (wrong "Incorrect arity") ) )) ) ) ) )
```

6.2.5. Аппликация

Единственная хоть сколь-нибудь сложная вещь — это аппликация, так как именно в этот момент решается, надо ли восстанавливать окружение или нет.

```
(define (meaning-regular-application e e* r tail?)
  (let* ((m (meaning e r #f))
         (m* (meaning* e* r (length e*) #f)) )
    (if tail? (lambda (k)
                (m (lambda (f)
                     (if (procedure? f)
                         (m* (lambda (v*)
                               (f v* k) ))              ; не трогаем
                         (wrong "Not a function" f) ) )) )
        (lambda (k)
          (m (lambda (f)
               (if (procedure? f)
                   (m* (lambda (v*)
                         (let ((sr *env*))              ; сохраняем
                           (f v* (lambda (v)
                                   (set! *env* sr)      ; возвращаем
                                   (k v) )) ) ))
                   (wrong "Not a function" f) ) )) ) ) ) )

(define (meaning* e* r size tail?)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size tail?)
      (meaning-no-arguments r size tail?) ) )

(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (index (- size (+ (length e*) 1))) )
    (lambda (k)
      (m (lambda (v)
           (m* (lambda (v*)
                 (set-activation-frame-argument! v* index v)
                 (k v*) )) )) ) ) )

(define (meaning-no-arguments r size tail?)
  (let ((size+1 (+ 1 size)))
    (lambda (k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

Таким образом, мы получаем интерпретатор с окружением, передаваемым через регистр, и оптимизацией хвостовых вызовов. Начальное состояние глобального окружения остаётся неизменным.

6.2.6. Выводы: интерпретатор с окружением в регистре

Такое преобразование не всегда является полезным. Например, если мы захотим добавить в язык параллелизм, то возникают соответствующие проблемы синхронизации: глобальная переменная `*env*` общая для всех потоков, но ведь у каждого из них своё личное окружение. Тем не менее, постоянно доступное окружение очень полезно, например, для рефлексии.

Новый интерпретатор ничуть не быстрее предыдущего. Да, функции теперь имеют на один аргумент меньше, но ценой постоянных обращений к изменяемой глобальной переменной, значение которой предсказывать сложнее. Однако, мы всё же прояснили немаловажное понятие хвостовых вызовов, а также рассмотрели несколько других вопросов, которые пригодятся в следующем интерпретаторе.

6.3. Укрощение продолжений

Язык, используемый для реализации, зачастую предоставляет некоторый доступ к продолжениям напрямую, что избавляет нас от необходимости явной работы с ними. Использование данных возможностей — это вполне разумный подход. Наличие готового компилятора Scheme значительно облегчает написание интерпретатора: достаточно написать его на Scheme и откомпилировать, при этом `call/cc` и другие встроенные функции просто берутся из библиотеки времени исполнения этого компилятора. Для простого кода на Лиспе вообще будет достаточно примитивов `setjmp`/`longjmp`, присутствующих в библиотеке языка Си. В обоих случаях в определяемом языке `call/cc` становится магической данностью, что позволяет призывать продолжения лишь тогда, когда они действительно необходимы. Без `call/cc`, спущенной свыше, мы обязаны постоянно создавать продолжения самостоятельно, просто потому, что они *могут* понадобиться. Очевидно, это сказывается на производительности интерпретатора.

Следующий интерпретатор, соответственно, не будет использовать явные продолжения. Параллельно с этим улучшением будут реализованы ещё два других:

1) Явное представление функций с помощью объектов.

2) Результатом предварительной обработки станут комбинаторы (чьи имена записываются ПРОПИСНЫМИ БУКВАМИ), напоминающие инструкции гипотетической виртуальной машины.

Естественно, все оптимизации из предыдущего раздела (встраивание примитивов, приводимые формы и т. д.) останутся, разве что будут переписаны на новый лад.

6.3.1. Замыкания

Замыкания представляются так же, как и ранее в третьей главе, — объектами с двумя полями: одно для кода, другое для окружения определения. Для вызова таких замыканий вводится специальная функция `invoke`.

```
(define-class closure Object
  ( code
    closed-environment ) )

(define (invoke f v*)
  (if (closure? f)
      ((closure-code f) v* (closure-closed-environment f))
      (wrong "Not a function" f) ) )
```

Интерпретируемый код функций представляется замыканиями, принимающими два аргумента: запись активации и некоторое окружение. Благодаря этому мы сможем при надобности расширять окружение определения дополнительными переменными. Каждое замыкание, соответственно, должно вызываться исключительно с помощью `invoke`.

6.3.2. Предобработчик

Предварительная обработка программ выполняется всё той же функцией `meaning`. Однако, теперь вместо `(lambda (k) ...)` она возвращает замыкания вида `(lambda () ...)`, которые можно понимать как адреса, на которые достаточно просто перейти для «вызова». (Forth язык в корнями способ уходит подобный.) Фактически, мы получаем обещание исполнения обработанной программы. Более того, если немного видоизменить протокол вызова функций, то для выполнения данных обещаний можно реализовать очень эффективный оператор `GOTO`. [**см. упр. 6.6**]

Определение `meaning` идентично приведённому на странице 250.

6.3.3. Цитирование

Цитаты остаются неизменными, но теперь их определение более читабельно благодаря комбинатору `CONSTANT`:

```
(define (meaning-quotation v r tail?)
  (CONSTANT v) )

(define (CONSTANT value)
  (lambda () value) )
```

6.3.4. Обращения к переменным

Обработка переменных сводится к определению их класса и вызову соответствующих функций-аксессоров. Данные аксессоры являются комбинаторами, что повышает удобочитаемость и понятность.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (CHECKED-GLOBAL-REF i) ) )
          ((predefined)
           (let ((i (cdr kind)))
             (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )

(define (SHALLOW-ARGUMENT-REF j)
  (lambda () (activation-frame-argument *env* j)) )

(define (PREDEFINED i)
  (lambda () (predefined-fetch i)) )

(define (DEEP-ARGUMENT-REF i j)
  (lambda () (deep-fetch *env* i j)) )

(define (GLOBAL-REF i)
  (lambda () (global-fetch i)) )

(define (CHECKED-GLOBAL-REF i)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable")
          v ) ) ) )
```

Заметьте, что в случае, если переменная окажется неинициализированной, комбинатор `CHECKED-GLOBAL-REF` будет иметь в распоряжении только её адрес в `sg.current`, так что красивое сообщение об ошибке с именем переменной вывести не получится. Чтобы иметь возможность так делать, во время исполнения программы необходимо хранить то, что называется *таблицей символов*:

структуру данных, устанавливающую соответствия между адресами и именами переменных (во время предобработки ей соответствует список `g.current`). [**см. упр. 6.1**] Тогда в случае нахождения дефектной переменной мы сможем выдать и её имя, а не только непонятный адрес.

6.3.5. Ветвление

Условный оператор тоже становится гораздо яснее благодаря комбинатору `ALTERNATIVE`, принимающему результаты предварительной обработки всех трёх подформ оператора.

```
(define (meaning-alternative e1 e2 e3 r tail?)
  (let ((m1 (meaning e1 r #f))
        (m2 (meaning e2 r tail?))
        (m3 (meaning e3 r tail?)) )
    (ALTERNATIVE m1 m2 m3) ) )

(define (ALTERNATIVE m1 m2 m3)
  (lambda ()
    (if (m1) (m2) (m3))) )
```

6.3.6. Присваивание

Присваивание нового значения переменной напоминает получение её текущего значения, за исключением того, что здесь ещё надо вычислить дополнительную форму. Эта подформа будет передаваться во все нуждающиеся в ней комбинаторы.

```
(define (meaning-assignment n e r tail?)
  (let ((m (meaning e r #f))
        (kind (compute-kind r n)) )
    (if kind
        (case (car kind)
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-SET! j m)
                 (DEEP-ARGUMENT-SET! i j m) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (GLOBAL-SET! i m) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (lambda () (set-activation-frame-argument! *env* j (m))) )

(define (DEEP-ARGUMENT-SET! i j m)
  (lambda () (deep-update! *env* i j (m))) )

(define (GLOBAL-SET! i m)
  (lambda () (global-update! i (m))) )
```

6.3.7. Последовательность

Последовательное вычисление форм выражается с помощью комбинатора SEQUENCE, являющегося бинарным аналогом begin.

```
(define (meaning-sequence e+ r tail?)
  (if (pair? e+)
      (if (pair? (cdr e+))
          (meaning*-multiple-sequence (car e+) (cdr e+) r tail?)
          (meaning*-single-sequence (car e+) r tail?) )
      (static-wrong "Illegal syntax: (begin)") ) )

(define (meaning*-single-sequence e r tail?)
  (meaning e r tail?) )

(define (meaning*-multiple-sequence e e+ r tail?)
  (let ((m1 (meaning e r #f))
        (m+ (meaning-sequence e+ r tail?)) )
    (SEQUENCE m1 m+) ) )

(define (SEQUENCE m m+)
  (lambda () (m) (m+)) )
```

6.3.8. Абстракция

Замыкания создают комбинаторы FIX-CLOSURE и NARY-CLOSURE. Различаются они предикатами, проверяющими арность, а также подходами к формированию списка аргументов.

```
(define (meaning-abstraction nn* e+ r tail?)
  (let parse ((n* nn*)
              (regular '()) )
    (cond
     ((pair? n*) (parse (cdr n*) (cons (car n*) regular)))
     ((null? n*) (meaning-fix-abstraction nn* e+ r tail?))
     (else       (meaning-dotted-abstraction
                  (reverse regular) n* e+ r tail? )) ) ) )
```

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (FIX-CLOSURE m+ arity) ) )

(define (meaning-dotted-abstraction n* n e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence e+ r2 #t)) )
    (NARY-CLOSURE m+ arity) ) )

(define (FIX-CLOSURE m+ arity)
  (let ((arity+1 (+ 1 arity)))
    (lambda ()
      (define (the-function v* sr)
        (if (= (activation-frame-argument-length v*) arity+1)
            (begin (set! *env* (sr-extend* sr v*))
                   (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )

(define (NARY-CLOSURE m+ arity)
  (let ((arity+1 (+ 1 arity)))
    (lambda ()
      (define (the-function v* sr)
        (if (>= (activation-frame-argument-length v*) arity+1)
            (begin
              (listify! v* arity)
              (set! *env* (sr-extend* sr v*))
              (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
```

6.3.9. Аппликация

Осталось разобраться только с вызовами функций. `meaning-application` анализирует форму аппликации и умеет определять приводимые формы, обращения к примитивам и все виды обычных аппликаций.

```
(define (meaning-application e e* r tail?)
  (cond ((and (symbol? e)
              (let ((kind (compute-kind r e)))
                (and (pair? kind)
                     (eq? 'predefined (car kind))
```

```
                    (let ((desc (get-description e)))
                      (and desc
                           (eq? 'function (car desc))
                           (or (= (length (cddr desc)) (length e*))
                               (static-wrong
                                "Incorrect arity for primitive" e )
                               ) ) ) ) ) )
          (meaning-primitive-application e e* r tail?) )
         ((and (pair? e)
               (eq? 'lambda (car e)) )
          (meaning-closed-application e e* r tail?) )
         (else (meaning-regular-application e e* r tail?)) ) )
```

Все обычные аппликации обрабатываются с помощью четырёх комбинаторов. Термы вычисляются как всегда слева направо.

```
(define (meaning-regular-application e e* r tail?)
  (let* ((m (meaning e r #f))
         (m* (meaning* e* r (length e*) #f)) )
    (if tail? (TR-REGULAR-CALL m m*) (REGULAR-CALL m m*)) ) )

(define (meaning* e* r size tail?)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size tail?)
      (meaning-no-arguments r size tail?) ) )

(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (index (- size (+ (length e*) 1))) )
    (STORE-ARGUMENT m m* index) ) )

(define (meaning-no-arguments r size tail?)
  (ALLOCATE-FRAME size) )

(define (TR-REGULAR-CALL m m*)
  (lambda ()
    (let ((f (m)))
      (invoke f (m*)) ) ) )

(define (REGULAR-CALL m m*)
  (lambda ()
    (let* ((f  (m))
           (v* (m*))
           (sr *env*)
           (result (invoke f v*)) )
      (set! *env* sr)
      result ) ) )
```

```
(define (STORE-ARGUMENT m m* index)
  (lambda ()
    (let* ((v  (m))
           (v* (m*)) )
      (set-activation-frame-argument! v* index v)
      v* ) ) )

(define (ALLOCATE-FRAME size)
  (let ((size+1 (+ 1 size)))
    (lambda ()
      (allocate-activation-frame size+1) ) ) )
```

6.3.10. Приводимые формы

Для обработки явно указанных в аппликациях `lambda`-форм потребуется четыре новых комбинатора. `CONS-ARGUMENT` собирает избыточные аргументы в список. `ALLOCATE-DOTTED-FRAME` создаёт запись активации, аналогичную создаваемой `ALLOCATE-FRAME`, но с последним элементом, равным `()` (тогда как `ALLOCATE-FRAME` не выполняет инициализацию из соображений производительности).

```
(define (meaning-dotted-closed-application n* n body e* r tail?)
  (let* ((m* (meaning-dotted* e* r (length e*) (length n*) #f))
         (r2 (r-extend* r (append n* (list n))))
         (m+ (meaning-sequence body r2 tail?)) )
    (if tail? (TR-FIX-LET m* m+)
              (FIX-LET m* m+) ) ) )

(define (meaning-dotted* e* r size arity tail?)
  (if (pair? e*)
      (meaning-some-dotted-arguments (car e*) (cdr e*)
                                     r size arity tail? )
      (meaning-no-dotted-arguments r size arity tail?) ) )

(define (meaning-some-dotted-arguments e e* r size arity tail?)
  (let ((m (meaning e r #f))
        (m* (meaning-dotted* e* r size arity tail?))
        (index (- size (+ (length e*) 1))) )
    (if (< index arity)
        (STORE-ARGUMENT m m* index)
        (CONS-ARGUMENT m m* arity) ) ) )

(define (meaning-no-dotted-arguments r size arity tail?)
  (ALLOCATE-DOTTED-FRAME arity) )
```

```
(define (FIX-LET m* m+)
  (lambda ()
    (set! *env* (sr-extend* *env* (m*)))
    (let ((result (m+)))
      (set! *env* (environment-next *env*))
      result ) ) )

(define (TR-FIX-LET m* m+)
  (lambda ()
    (set! *env* (sr-extend* *env* (m*)))
    (m+) ) )

(define (CONS-ARGUMENT m m* arity)
  (lambda ()
    (let* ((v  (m))
           (v* (m*)) )
      (set-activation-frame-argument! v* arity
       (cons v (activation-frame-argument v* arity)) )
      v* ) ) )

(define (ALLOCATE-DOTTED-FRAME arity)
  (let ((arity+1 (+ 1 arity)))
    (lambda ()
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        v* ) ) ) )
```

Комбинатор `FIX-LET` должен восстанавливать текущее окружение — а это значит, что оно должно быть где-то сохранено, чтобы его можно было оттуда восстановить. К счастью, есть элегантное решение: так как записи активаций связаны, то по самой верхней записи (бывшей текущей) можно легко определить предыдущую.

6.3.11. Примитивы

Осталось рассмотреть последний случай (по порядку, отнюдь не по важности): когда функциональным термом аппликации является имя глобальной неизменяемой переменной. В таком случае протокол вызова функций игнорируется, запись активации не создаётся и `invoke` не вызывается. Аппликация просто заменяется прямым вызовов нужного примитива.

```
(define (meaning-primitive-application e e* r tail?)
  (let* ((desc (get-description e))
         ;; desc = (function адрес . аргументы)
         (address (cadr desc))
         (size (length e*)) )
    (case size
```

```
        ((0) (CALL0 address))
        ((1)
         (let ((m1 (meaning (car e*) r #f)))
           (CALL1 address m1) ) )
        ((2)
         (let ((m1 (meaning (car e*) r #f))
               (m2 (meaning (cadr e*) r #f)) )
          (CALL2 address m1 m2) ) )
        ((3)
         (let ((m1 (meaning (car e*) r #f))
               (m2 (meaning (cadr e*) r #f))
               (m3 (meaning (caddr e*) r #f)) )
          (CALL3 address m1 m2 m3) ) )
        (else
         (meaning-regular-application e e* r tail?) ) ) ) )

(define (CALL0 address)
  (lambda () (address)) )

(define (CALL3 address m1 m2 m3)
  (lambda () (let* ((v1 (m1))
                    (v2 (m2))
                    (v3 (m3)) )
               (address v1 v2 v3) )) )
```

Аргументы `CALL3` явно вычисляются слева направо, чтобы не нарушать ранее установленный порядок.

6.3.12. Запускаем интерпретатор

Так как продолжения теперь присутствуют в интерпретаторе неявно, то структура определения примитивов слегка изменилась. Здесь показан только новый вариант макроса `defprimitive2`:

```
(define-syntax defprimitive2
  (syntax-rules ()
    ((defprimitive2 name value)
     (definitial name
       (letrec ((arity+1 (+ 2 1))
                (behavior
                 (lambda (v* sr)
                   (if (= arity+1 (activation-frame-argument-length v*))
                       (value (activation-frame-argument v* 0)
                              (activation-frame-argument v* 1) )
                       (wrong "Incorrect arity" 'name) ) ) ) )
         (description-extend! 'name `(function ,value a b))
         (make-closure behavior sr.init) ) ) ) ) )
```

Запускается интерпретатор следующим образом:

```
(define (chapter6.3-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    (display ((meaning (read) r.init #t)))
    (toplevel) )
  (toplevel) )
```

6.3.13. Функция `call/cc`

`call/cc` в этом интерпретаторе является магией, так что для её определения необходима `call/cc` из библиотеки языка реализации. Это возвращает нас к тавтологическим определениям из первых глав.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-closure
      (lambda (v* sr)
        (if (= arity+1 (activation-frame-argument-length v*))
            (call/cc
             (lambda (k)
               (invoke
                (activation-frame-argument v* 0)
                (let ((frame (allocate-activation-frame (+ 1 1))))
                  (set-activation-frame-argument!
                   frame 0
                   (make-closure
                    (lambda (values r)
                      (if (= (activation-frame-argument-length values)
                             arity+1 )
                          (k (activation-frame-argument values 0))
                          (wrong "Incorrect arity" 'continuation) ) )
                    sr.init ) )
                  frame ) ) ) )
            (wrong "Incorrect arity" 'call/cc) ) )
      sr.init ) ) )
```

6.3.14. Функция `apply`

Выходя за пределы обычного круга обсуждаемых вопросов, давайте определим функцию `apply`. Её всегда довольно непросто написать, так как она сильно зависит от представления функций и протокола их вызова. Однако, этим нас не испугать.

```
(definitial apply
  (let* ((arity 2)
         (arity+1 (+ 1 arity)) )
    (make-closure
     (lambda (v* sr)
       (if (>= (activation-frame-argument-length v*) arity+1)
           (let* ((proc (activation-frame-argument v* 0))
                  (last-arg-index
                   (- (activation-frame-argument-length v*) 2) )
                  (last-arg
                   (activation-frame-argument v* last-arg-index) )
                  (size (+ last-arg-index (length last-arg)))
                  (frame (allocate-activation-frame size)) )
             (do ((i 1 (+ i 1)))
                 ((= i last-arg-index))
               (set-activation-frame-argument!
                frame (- i 1) (activation-frame-argument v* i) ) )
             (do ((i (- last-arg-index 1) (+ i 1))
                  (last-arg last-arg (cdr last-arg)) )
                 ((null? last-arg))
               (set-activation-frame-argument! frame i (car last-arg)) )
             (invoke proc frame) )
         (wrong "Incorrect arity" 'apply) ) )
     sr.init ) ) )
```

Этот примитив сначала проверяет, что ему передано по крайней мере два аргумента, затем проходит по записи своей активации, чтобы выяснить точное количество аргументов, передаваемых вызываемой функции. Для этого необходимо учесть длину списка, хранимого в последнем элементе записи активации `apply`. Как только у `apply` есть это число, можно создать запись активации правильного размера для вызываемой функции. Затем обычные аргументы переносятся напрямую из одной записи в другую, а список «лишних» аргументов приходится распарывать и обрабатывать поэлементно. Условие окончания списка проверяется с помощью `null?`. Использование `atom?` кажется более здравым, но это бы сделало программу `(apply list '(a b . c))` допустимой — вопреки требованиям стандарта.

Как видим, `apply` — это отнюдь не дешёвая операция, так как она вынуждена создавать новую запись активации, а также проходить по списку, хранимому в точечном аргументе.

6.3.15. Выводы: интерпретатор без продолжений

Новый интерпретатор приблизительно от двух до четырёх раз быстрее предыдущего, в основном из-за отсутствия постоянно создаваемых замыканий с продолжениями. Представление продолжений замыканиями не учитывает их важной особенности: обычно продолжения не живут так же долго,

как остальные значения. Эта особенность позволяет очень эффективно размещать продолжения в аппаратном стеке, а также значительно облегчает удаление ненужных продолжений: сдвинуть верхушку стека можно за одну-единственную машинную инструкцию. Дабы не вводить вас в заблуждение, поясним эту мысль: количество памяти, необходимое для размещения продолжений, не зависит от того, будут ли они жить в стеке или в куче, но благодаря обычно небольшому времени жизни продолжений их эффективнее размещать именно на стеке. (В [App87] есть иное мнение на этот счёт.) Как показано в [MB93], данное утверждение остаётся верным, даже если использовать отдельную кучу специально для продолжений.

Способ использования обещаний, показанный в этой главе, известен как *шитый код* [Bel73] и широко применяется при реализации языка Forth [Hon93]. Вдохновение при создании этого интерпретатора также черпалось из работы Марка Филе и Гая Лапальма: [FL87].

Комбинаторы фактически играют роль кодогенераторов. Они используются как простой интерфейс представления кода, позволяющий легко изменять реализацию исполнителя предобработанных программ. С их помощью довольно просто начать наконец представлять код функций настоящими объектами, а не замыканиями. По сути, компиляция, которую мы будем рассматривать в следующей главе, является лишь серьёзной доработкой одного из упражнений: [**см. упр. 6.3**].

6.4. Заключение

На первый взгляд, вся эта глава и её три интерпретатора — это огромный шаг назад, лишивший нас всего приобретённого в первых четырёх. Действительно, память практически исчезла (сократившись до управления записями активаций), продолжений нет совсем. С другой стороны, мы подробно разобрали предварительную обработку программ, которая является важным этапом компиляции. Кроме того, вычисления были разделены на статические и динамические, что позволило реализовать множество оптимизаций, ускоряющих интерпретацию. И мы действительно её ускорили — примерно на два порядка по сравнению с денотационным интерпретатором.

Третий интерпретатор, рассмотренный в данной главе, реализует все специальные формы Scheme и представляет собой квинтэссенцию практически любого языка. Остаётся только снабдить его менеджером памяти, а также библиотеками для работы с текстом, графикой, моделирования, физических расчётов, прототипирования языков, виртуальной реальности и т. д. Конечно, нам легко сказать: «только», а на самом деле за этим словом скрывается неимоверная сложность разработки удобного способа представления примитивных объектов в памяти, который бы обеспечил возможность быстрой проверки типов [Gud93] и эффективной сборки мусора, не говоря уже о как минимум равной трудоёмкости задачи написания прикладных библиотек.

Естественно, мы могли бы и дальше улучшать приведённые здесь интерпретаторы или, например, расширить их другими специальными формами, но помните, что их главной задачей является иллюстрация инкрементального подхода к разработке, позволяющего сделать описание языка более компактным и понятным, а также формирующего фундамент для нашего дальнейшего продвижения.

6.5. Упражнения

Упражнение 6.1 Доработайте комбинатор `CHECKED-GLOBAL-REF`, научив его выводить более осмысленные сообщения о неинициализированных переменных. [**см. стр. 256**]

Упражнение 6.2 Определите примитив `list` для третьего интерпретатора из этой главы. Подсказка: это можно сделать *очень* элегантно.

Упражнение 6.3 Вместо исполнения программы мы могли бы выводить на экран результат предварительной обработки. Как-то так это бы выглядело для факториала:

```
? (disassemble
    '(lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) )

= (FIX-CLOSURE
    (ALTERNATIVE
      (CALL2 #<=> (SHALLOW-ARGUMENT-REF 0) (CONSTANT 0))
      (CONSTANT 1)
      (CALL2 #<*> (SHALLOW-ARGUMENT-REF 0)
                  (REGULAR-CALL
                    (CHECKED-GLOBAL-REF 10)  ; ← fact
                    (STORE-ARGUMENT
                      (CALL2 #<-> (SHALLOW-ARGUMENT-REF 0)
                                  (CONSTANT 1) )
                      (ALLOCATE-FRAME 1)
                      0 ) ) ) )
    1 )
```

Напишите подобный дизассемблер.

Упражнение 6.4 Измените последний рассмотренный интерпретатор так, чтобы записи активаций создавались перед вычислением аргументов. Тогда готовые аргументы можно будет сразу же укладывать в ячейки записи активации. [**см. стр. 229**]

Упражнение 6.5 Определите специальную форму `redefine`, позволяющую переопределять глобальные неизменяемые переменные. Она должна принимать имя примитива и создавать одноименную переменную в глобальном изменяемом окружении, которая, как сказано на странице 231, скроет оригинал. Пусть начальным значением этой переменной будет переопределяемый примитив.

Упражнение 6.6 Оптимизируйте предобработку функций без аргументов. [**см. стр. 255**]

Рекомендуемая литература

Последний рассмотренный интерпретатор вдохновлён статьёй [FL87]. Идея перехода от денотаций к комбинаторам взята из [Cli84]. Если вам действительно понравилось оптимизировать интерпретацию, то определённо стоит изучить работы [Cha80] и [SJ93].

Глава 7

Компиляция

В предыдущей главе был изложен метод перевода программ со Scheme на древовидный язык, содержащий около двадцати инструкций. В этой главе мы покажем, как преобразовать результат данной предварительной обработки в последовательность байтов — специализированный машинный язык. По пути будут рассмотрены следующие темы: создание виртуальной машины, компиляция в её внутренний язык, реализация различных расширений Scheme вроде переходов, динамических переменных, исключений.

(SHALLOW-ARGUMENT-REF j)	(PREDEFINED i)
(DEEP-ARGUMENT-REF i j)	(SHALLOW-ARGUMENT-SET! j m)
(DEEP-ARGUMENT-SET! i j m)	(GLOBAL-REF i)
(CHECKED-GLOBAL-REF i)	(GLOBAL-SET! i m)
(CONSTANT v)	(ALTERNATIVE m_1 m_2 m_3)
(SEQUENCE m $m+$)	(TR-FIX-LET $m*$ $m+$)
(FIX-LET $m*$ $m+$)	(CALL0 *адрес*)
(CALL1 *адрес* m_1)	(CALL2 *адрес* m_1 m_2)
(CALL3 *адрес* m_1 m_2 m_3)	(FIX-CLOSURE $m+$ *арность*)
(NARY-CLOSURE $m+$ *арность*)	(TR-REGULAR-CALL m $m*$)
(REGULAR-CALL m $m*$)	(STORE-ARGUMENT m $m*$ *индекс*)
(CONS-ARGUMENT m $m*$ *арность*)	(ALLOCATE-FRAME *размер*)
(ALLOCATE-DOTTED-FRAME *арность*)	

Таблица 7.1. Все 25 инструкций промежуточного языка. m, m_1, m_2, m_3, $m+$ и v — значения; $m*$ — запись активации; *индекс*, *арность*, *размер*, i и j — натуральные числа (с нулём); *адрес* представляет примитивную функцию, принимающую и возвращающую значения.

В результате компиляции обычно получается последовательность весьма низкоуровневых инструкций. Это отнюдь не так для нашего предобработчика: мы получаем на выходе структурированную древовидную программу. Рассмотрим следующий красноречивый пример. Пусть у нас есть программа:

```
((lambda (fact) (fact 5 fact (lambda (x) x)))
 (lambda (n f k) (if (= n 0) (k 1)
                     (f (- n 1) f (lambda (r) (k (* n r)))) )) )
```

После обработки она превращается в следующую:

```
(TR-FIX-LET
 (STORE-ARGUMENT
  (FIX-CLOSURE
   (ALTERNATIVE
    (CALL2 #<=> (SHALLOW-ARGUMENT-REF 0) (CONSTANT 0))
    (TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 2)
                     (STORE-ARGUMENT (CONSTANT 1)
                                     (ALLOCATE-FRAME 1) 0) )
    (TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 1)
     (STORE-ARGUMENT (CALL2 #<-> (SHALLOW-ARGUMENT-REF 0) (CONSTANT 1))
      (STORE-ARGUMENT (SHALLOW-ARGUMENT-REF 1)
       (STORE-ARGUMENT (FIX-CLOSURE
                        (TR-REGULAR-CALL (DEEP-ARGUMENT-REF 1 2)
                         (STORE-ARGUMENT (CALL2 #<*>
                                         (DEEP-ARGUMENT-REF 1 0)
                                         (SHALLOW-ARGUMENT-REF 0) )
                          (ALLOCATE-FRAME 1)
                          0 ) )
                        1 )
        (ALLOCATE-FRAME 3)
        2 )
       1 )
      0 ) ) )
   3 )
  (ALLOCATE-FRAME 1)
  0 )
 (TR-REGULAR-CALL (SHALLOW-ARGUMENT-REF 0)
  (STORE-ARGUMENT (CONSTANT 5)
   (STORE-ARGUMENT (SHALLOW-ARGUMENT-REF 0)
    (STORE-ARGUMENT (FIX-CLOSURE (SHALLOW-ARGUMENT-REF 0) 1)
     (ALLOCATE-FRAME 3)
     2 )
    1 )
   0 ) ) )
```

Не шедевр удобочитаемости, но очень точно выражает необходимые действия. Цель данной главы — показать, что данная форма ещё далека от окончательной. Даже не форма — после выполнения преобразований вроде линеаризации и байт-кодирования мы получим совершенно новый язык. Тот же язык, инструкции-генераторы которого собраны в таблице 7.1, на самом деле будет служить нам лишь промежуточным этапом, трамплином, от которого мы оттолкнёмся, чтобы улететь в дальние миры.

Предварительная обработка (выполняемая третьим интерпретатором из предыдущей главы) будет первым проходом компилятора. Соответственно, компилятор должен понимать лишь эти двадцать пять инструкций. На самом

деле, мы их даже немного улучшим, заточив под целевой язык виртуальной машины. Подобное разделение труда возможно благодаря тому, что промежуточный язык уже является исполнимым. Это позволяет протестировать предобработчик отдельно и полностью сконцентрироваться на второй фазе компиляции.

Для начала мы займёмся собственно компиляцией под виртуальную машину. Это будет простая, но типичная виртуальная машина, программируемая с помощью машинного языка. Её инструкции представляются байтами, целыми числами от 0 до 255. Такой способ представления программ называется *байт-кодом*. Изобретён он был незадолго до 1980 года, судя по [Deu80, Row80]. С тех пор байт-код нередко используется для иллюстрации процесса компиляции; как пример: [Hen80]. Получаемый код является довольно компактным — качество, очень полезное на машинах с ограниченной памятью или кешем. Именно компиляция в байт-код применяется для PC-Scheme [BJ86] и Caml Light [LW93].

Концептуально байт-кодирование устроено довольно просто: прошедшая предобработку программа компилируется в последовательность байт-кодов, которая уже исполняется интерпретатором, эмулирующим виртуальную машину на реальной. Так как байт-коды проще, чем высокоуровневые языки, то интерпретатор тратит значительно меньше времени на их разбор и в итоге работает быстрее.

7.1. Компиляция в байт-код

Задача на сейчас: шаг за шагом разработать специализированную машину, исполняющую байт-коды. Мы определим эту машину, определив для неё смысл (операционную семантику) двадцати пяти инструкций промежуточного языка. Часть из них трактуются очевидным образом, другие потребуют некоторой изобретательности. К счастью, виртуальная машина и её язык создаются одновременно, что даёт неоценимую гибкость разработки. Для нас не будет проблемой в любой момент добавить ещё один регистр или, например, дополнить архитектуру стеком.

Чтобы получить желаемую последовательность байт-кодов, потребуется вытянуть в прямую линию изначальное дерево инструкций промежуточного языка, *линеаризовать* его. В данный момент эти инструкции обмениваются информацией через аргументы и возвращаемые значения, но возможности передачи информации между машинным инструкциями ограничены регистрами и стеком. Сейчас наша машина имеет лишь один регистр: `*env*`, содержащий текущее состояние лексического окружения, но вскоре эта слегка спартанская архитектура обрастёт жирком.

7.1.1. Знакомьтесь, регистр *val*

Некоторые инструкции промежуточного языка производят значения, например, SHALLOW-ARGUMENT-REF или CONSTANT. Другие же, подобные FIX-LET или ALTERNATIVE, только управляют ходом вычислений. Помимо них есть и третья разновидность инструкций, которые потребляют значения, производимые первыми. Давайте внимательно рассмотрим инструкцию GLOBAL-SET!, определяемую следующим образом:

```
(define (GLOBAL-SET! i m)
  (lambda () (global-update! i (m))) )
```

Для того, чтобы передать инструкции global-update! результат вычисления (m), необходим дополнительный регистр. Назовём его *val*. Инструкции-производители, соответственно, кладут возвращаемые значения в этот регистр, а инструкции-потребители — забирают их оттуда. В итоге, типичный производитель CONSTANT теперь записывается вот так:

```
(define (CONSTANT value)
  (lambda ()
    (set! *val* value) ) )
```

а потребители значений вроде GLOBAL-SET! становятся такими:

```
(define (GLOBAL-SET! i m)
  (lambda ()
    (m)
    (global-update! i *val*) ) )
```

Итого, форма (m) размещает новое значение переменной в регистре *val*, затем его оттуда забирает global-update! и переносит в глобальное окружение. Заметьте, что global-update! не изменяет значение регистра *val*, ведь это потребовало бы как минимум одной лишней инструкции. Следовательно, возвращаемым значением этой формы присваивания является только что присвоенное глобальной переменной значение.

По этому примеру довольно легко догадаться, как преобразовать остальные инструкции. Например, SEQUENCE вообще не требует изменений:

```
(define (SEQUENCE m m+)
  (lambda () (m) (m+)) )
```

а FIX-LET принимает следующий вид:

```
(define (FIX-LET m* m+)
  (lambda ()
    (m*)
    (set! *env* (sr-extend* *env* *val*))
    (m+)
    (set! *env* (activation-frame-next *env*)) ) )
```

7.1.2. Изобретение стека

На данный момент часть инструкций уже линеаризована с помощью регистра `*val*`, но некоторые из них так просто не поддаются. Например, `STORE-ARGUMENT` сейчас имеет такое определение:

```
(define (STORE-ARGUMENT m m* index)
  (lambda ()
    (m)
    (let ((v *val*))
      (m*)
      (set-activation-frame-argument! *val* index v) ) ) )
```

Эта инструкция использует `let`, чтобы сохранить, а затем восстановить значение регистра `*val*`. Форма `let` сохраняет значение в «анонимном регистре» `v` на время, пока вычисляется `(m*)`. Под `v` не получится выделить реальный машинный регистр, так как нам может потребоваться произвольное количество таких `v` одновременно — ведь, например, внутри `m*` вполне может быть ещё несколько вложенных `STORE-ARGUMENT`. Следовательно, здесь необходимо нечто, где можно хранить несколько значений между инструкциями. Для этой цели уместно будет использовать стек, так как он прекрасно отображает идею сбалансированных сохранений-восстановлений. Определяется он элементарно:

```
(define *stack* (make-vector 1000))
(define *stack-index* 0)

(define (stack-push v)
  (vector-set! *stack* *stack-index* v)
  (set! *stack-index* (+ *stack-index* 1)) )

(define (stack-pop)
  (set! *stack-index* (- *stack-index* 1))
  (vector-ref *stack* *stack-index*) )
```

Обретя данную революционную технологию, мы тут же используем её для радикального преобразования инструкции `STORE-ARGUMENT`. Теперь временные значения можно сохранить в стеке перед началом других вычислений, чтобы после их окончания достать оттуда всё в целости и сохранности. Правда, это сработает только при условии, что форма `(m*)` оставит стек в том же состоянии, в каком он был до её исполнения. Следовательно, необходимо быть аккуратным и везде следить за соблюдением данного инварианта.

```
(define (STORE-ARGUMENT m m* index)
  (lambda ()
    (m)
    (stack-push *val*)
    (m*)
    (set-activation-frame-argument! *val* index (stack-pop)) ) )
```

Модификация REGULAR-CALL аналогична, только здесь требуется хранить сразу два значения: саму функцию на время вычисления аргументов, а также текущее окружение на время выполнения функции. Сейчас эта инструкция выглядит так:

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (let ((f *val*))
      (m*)
      (let ((sr *env*))
        (invoke f *val*)
        (set! *env* sr) ) ) ) )
```

После добавления нового регистра для функций — *fun*, — определение преобразуется в следующее:

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (stack-push *val*)
    (m*)
    (set! *fun* (stack-pop))
    (stack-push *env*)
    (invoke *fun*)
    (set! *env* (stack-pop)) ) )
```

Заигравшись, мы мимоходом изменили протокол вызова функций. Теперь функции принимают свою запись активации не через аргумент, а ожидают её в регистре *val*. Это поведение закрепляется в определении FIX-CLOSURE:

```
(define (FIX-CLOSURE m+ arity)
  (let ((arity+1 (+ 1 arity)))
    (lambda ()
      (define (the-function sr)
        (if (= (activation-frame-argument-length *val*) arity+1)
            (begin (set! *env* (sr-extend* sr *val*))
                   (m+) )
            (wrong "Incorrect arity") ) )
      (set! *val* (make-closure the-function *env*)) ) ) )
```

Добавив ещё регистров, мы сможем линеаризовать и вызовы примитивов. Введём два новых регистра: *arg1* и *arg2*. Один из них может физически совпадать с *fun*, который всё равно не используется одновременно с ними. Эти регистры позволяет определить CALL3 следующим образом:

```
(define (CALL3 address m1 m2 m3)
  (lambda ()
    (m1)
    (stack-push *val*)
```

```
(m2)
(stack-push *val*)
(m3)
(set! *arg2* (stack-pop))
(set! *arg1* (stack-pop))
(set! *val* (address *arg1* *arg2* *val*)) ) )
```

7.1.3. Дорабатываем инструкции

В данный момент рассмотренные двадцать пять инструкций генерируют замыкания, внутри которых находятся последовательности составных операций над регистрами. Такие сложные инструкции нам не подходят — машине нужны маленькие и однозначные инструкции, выполняющие простые действия: изменить один регистр, положить одно значение в стек и т. п. Поэтому мы развернём большие инструкции, превратив их в последовательности из маленьких. Так как в такой куче команд легко запутаться, то давайте введём специализированный регистр для хранения следующей инструкции — *счётчик команд*. Заодно он позволит нам в дальнейшем точнее определить протокол вызова функций, а также всевозможные управляющие формы, в том числе и `call/cc`.

Линеаризация присваиваний

Рассмотрим для примера инструкцию `SHALLOW-ARGUMENT-SET!`. Определялась она вот так:

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (lambda ()
    (m)
    (set-activation-frame-argument! *env* j *val*) ) )
```

Чтобы разбить её на последовательность операций, введём вспомогательную функцию:

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (append m (SET-SHALLOW-ARGUMENT! j)) )

(define (SET-SHALLOW-ARGUMENT! j)
  (list (lambda () (set-activation-frame-argument! *env* j *val*))) )
```

Главная функция только возвращает список необходимых операций, расположенных в правильной последовательности. Вспомогательная же возвращает замыкание, выполняющее простую операцию изменения записи активации.

Линеаризация вызовов функций

REGULAR-CALL является весьма показательной инструкцией в этом плане. Для её линеаризации потребуются следующие машинные команды: PRESERVE-ENV, RESTORE-ENV, PUSH-VALUE, POP-FUNCTION и FUNCTION-INVOKE.

Вот их определения:

```
(define (REGULAR-CALL m m*)
  (append m (PUSH-VALUE)
          m* (POP-FUNCTION) (PRESERVE-ENV)
             (FUNCTION-INVOKE) (RESTORE-ENV) ) )

(define (PUSH-VALUE)
  (list (lambda () (stack-push *val*))) )

(define (POP-FUNCTION)
  (list (lambda () (set! *fun* (stack-pop)))) )

(define (PRESERVE-ENV)
  (list (lambda () (stack-push *env*))) )

(define (FUNCTION-INVOKE)
  (list (lambda () (invoke *fun*))) )

(define (RESTORE-ENV)
  (list (lambda () (set! *env* (stack-pop)))) )
```

Как мы и хотели, результатом компиляции теперь является список элементарных машинных инструкций. Его, однако, нельзя непосредственно исполнить, для этого придётся написать исполнитель самостоятельно. Назовём его run.

```
(define (run)
  (let ((instruction (car *pc*)))
    (set! *pc* (cdr *pc*))
    (instruction)
    (run) ) )
```

Полученный в результате компиляции список команд должен быть помещён в регистр *pc* — это наш счётчик команд (program counter). Функция run олицетворяет процессор виртуальной машины, который считывает команду, сдвигает счётчик на следующую, исполняет считанную команду и повторяет всё сначала. Кстати, именно поэтому все машинные инструкции представляются замыканиями вида (lambda () ...).

Линеаризация ветвлений

Способ линеаризации ветвления не так очевиден, ведь из него возможны два выхода. Как же представить такое поведение последовательностью? Для

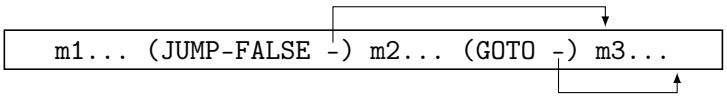


Рис. 7.1. Линеаризованное ветвление.

этого мы (пере)изобретём две важные машинные инструкции, позволяющие влиять на счётчик команд: `JUMP-FALSE` и `GOTO`. Всем известная `GOTO` выполняет безусловный переход. В свою очередь, `JUMP-FALSE` переход выполняет лишь в случае, когда в регистре `*val*` находится ложь. Обе эти инструкции изменяют исключительно `*pc*` и ничего другого.

```
(define (JUMP-FALSE i)
  (list (lambda () (if (not *val*) (set! *pc* (list-tail *pc* i))))) )

(define (GOTO i)
  (list (lambda () (set! *pc* (list-tail *pc* i)))) )
```

С использованием этих двух инструкций ветвление линеаризуется следующим образом. (На рисунке 7.1 показано нагляднее.)

```
(define (ALTERNATIVE m1 m2 m3)
  (append m1 (JUMP-FALSE (+ 1 (length m2))) m2 (GOTO (length m3)) m3) )
```

Условие вычисляется, а затем проверяется `JUMP-FALSE`. Если оно оказалось истиной, то выполняются инструкции, непосредственно следующие за `JUMP-FALSE`, после чего `GOTO` перебрасывает управление через альтернативную ветку к коду, следующему за ветвлением. Если же условие ложно, то управление просто передаётся альтернативной ветке. Как видим, для реализации ветвления нам пришлось опуститься до уровня ассемблера. Кстати, обратите внимание, что переходы выполняются относительно текущего значения счётчика команд, а не по абсолютным адресам.

Линеаризация абстракций

Последней из инструкций с нетривиальной линеаризацией остаётся конструктор замыканий. Проблема здесь в том, как разместить код собственно функции и код, замыкающий текущее окружение. Причём функцию не надо сразу же исполнять после создания замыкания. Для реализации такого поведения снова используются переходы (см. рисунок 7.2). Вот так создаются замыкания с переменной арностью:

```
(define (NARY-CLOSURE m+ arity)
  (define the-function
    (append (ARITY>=? (+ arity 1)) (PACK-FRAME! arity)
            (EXTEND-ENV) m+ (RETURN) ) )
  (append (CREATE-CLOSURE 1) (GOTO (length the-function))
          the-function ) )
```

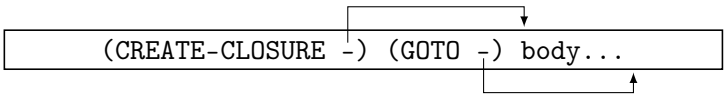


Рис. 7.2. Линеаризованная абстракция.

```
(define (CREATE-CLOSURE offset)
  (list (lambda () (set! *val* (make-closure (list-tail *pc* offset)
                                             *env* )))) )
(define (PACK-FRAME! arity)
  (list (lambda () (listify! *val* arity))) )
```

Новая машинная инструкция `CREATE-CLOSURE` создаёт замыкание с телом, расположенным сразу же за `GOTO`. Созданное замыкание кладётся в регистр `*val*`, после чего мы перепрыгиваем через его код к следующей инструкции программы.

7.1.4. Протокол вызова функций

Вызовы функций выполняются с помощью инструкций `TR-REGULAR-CALL` и `REGULAR-CALL`, рассмотренных ранее. [**см. стр. 276**] Функция `invoke` реализует протокол вызова функций:

```
(define (invoke f)
  (cond ((closure? f)
         (stack-push *pc*)
         (set! *env* (closure-closed-environment f))
         (set! *pc* (closure-code f)) )
        ... ) )
```

Перед вызовом функции в стек заталкивается адрес инструкции, следующей за `(FUNCTION-INVOKE)`. Затем извлекается сохранённое в замыкании окружение и устанавливается в регистр `*env*`. Наконец, в регистр `*pc*` заносится адрес первой инструкции тела замыкания, что передаёт ему управление. Текущее окружение здесь не сохраняется: за управление ним отвечают непосредственно `REGULAR-CALL` и `TR-REGULAR-CALL`.

Следующей инструкцией, исполненной `run`, будет первая инструкция вызванной функции. Она проверяет арность и, в случае успеха, расширяет текущее окружение записью активации функции, в тот момент уже находящейся в регистре `*val*`. Теперь всё готово для исполнения тела функции. После его завершения в регистре `*val*` будет находиться значение, которое требуется передать тому, кто вызывал эту функцию. Возврат управления выполняет инструкция `RETURN`. Ей надо лишь восстановить значение `*pc*`, ранее сохранённое в стеке.

```
(define (RETURN)
  (list (lambda () (set! *pc* (stack-pop)))) )
```

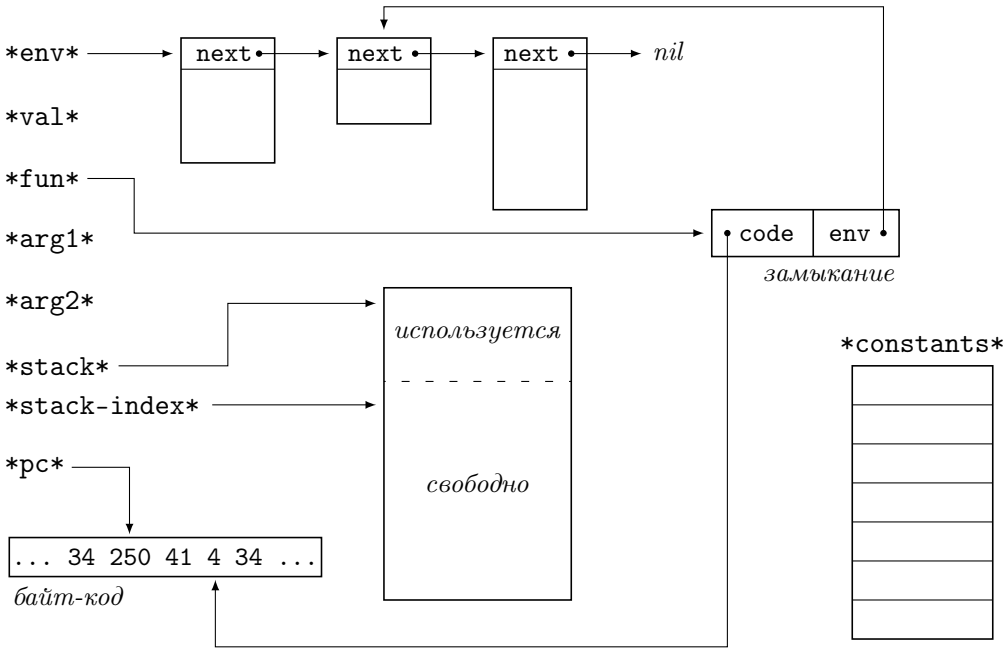


Рис. 7.3. Архитектура целевой виртуальной машины.

О переходах

Если вы знакомы с программированием на языке ассемблера, то наверняка обратили внимание на то, что все переходы выполняются только вперёд по коду. Более того, переходы являются относительными, равно как и адресация при создании замыканий. Следовательно, получаемый машинный код никак не зависит от своего фактического расположения в памяти, он *позиционно-независим*.

7.2. Язык и целевая машина

Пришло время зафиксировать архитектуру целевой виртуальной машины, под которую будут компилироваться программы, а также машинный язык, с помощью которого ей можно управлять. Как видно из рисунка 7.3, машина имеет пять регистров: `*env*`, `*val*`, `*fun*`, `*arg1*` и `*arg2*`, счётчик команд `*pc*`, а также стек.

Машинный язык состоит из тридцати четырёх инструкций. Все они собраны в таблице 7.2. Вдобавок к ранее рассмотренным, мы ввели инструкцию `FINISH`, которая завершает исполнение программы (фактически, выходит из функции `run`) и возвращает управление тому, кто запустил нашу виртуальную машину.

Все двадцать пять инструкций-генераторов промежуточного языка можно разделить на две группы: простые инструкции и составные. Простые инструкции имеют непосредственные аналоги в машинном языке, они отмечены звёздочками в таблице 7.2. В противоположность им, шестнадцать составных инструкций промежуточного языка определяются через оставшиеся двадцать пять машинных. Мы могли бы разбивать инструкции на машинные команды более агрессивно, например, разделив `CHECKED-GLOBAL-REF` на `GLOBAL-REF` и отдельную инструкцию, проверяющую, действительно ли в `*val*` лежит значение инициализированной переменной, но это бы замедлило интерпретацию. Так что скорее наоборот, некоторые инструкции имеет смысл объединить, например: `POP-ARG2` используется только в `CALL3`, причём за `POP-ARG2` всегда следует `POP-ARG1`, поэтому данную пару вполне можно превратить в отдельную инструкцию.

(SHALLOW-ARGUMENT-REF j)$_*$	(PREDEFINED i)$_*$
(DEEP-ARGUMENT-REF i j)$_*$	(SET-SHALLOW-ARGUMENT! j)
(SET-DEEP-ARGUMENT! i j)	(GLOBAL-REF i)$_*$
(CHECKED-GLOBAL-REF i)$_*$	(SET-GLOBAL! i)
(CONSTANT v)$_*$	(JUMP-FALSE *смещение*)
(GOTO *смещение*)	(EXTEND-ENV)
(UNLINK-ENV)	(CALL0 *адрес*)$_*$
(INVOKE1 *адрес*)	(PUSH-VALUE)
(INVOKE2 *адрес*)	(POP-ARG1)
(INVOKE3 *адрес*)	(POP-ARG2)
(CREATE-CLOSURE *смещение*)	(ARITY=? *арность* + 1)
(RETURN)	(ARITY>=? *арность* + 1)
(PACK-FRAME! *арность*)	(POP-FUNCTION)
(FUNCTION-INVOKE)	(PRESERVE-ENV)
(RESTORE-ENV)	(POP-FRAME! *индекс*)
(POP-CONS-FRAME! *арность*)	(ALLOCATE-FRAME *размер*)$_*$
(ALLOCATE-DOTTED-FRAME *арность*)$_*$	(FINISH)

Таблица 7.2. Символьная форма машинных инструкций.

Тривиальной части составных инструкций мы не касались, но их определения на машинном языке всё же стоит привести. Они собраны ниже, без каких-либо пояснений. Некоторые из машинных инструкций из таблицы 7.2 тоже не были определены, ими мы займёмся чуть позже.

```
(define (DEEP-ARGUMENT-SET! i j m)
  (append m (SET-DEEP-ARGUMENT! i j)) )

(define (GLOBAL-SET! i m)
  (append m (SET-GLOBAL! i)) )
```

```
(define (SEQUENCE m m+)
  (append m m+) )

(define (TR-FIX-LET m* m+)
  (append m* (EXTEND-ENV) m+) )

(define (FIX-LET m* m+)
  (append m* (EXTEND-ENV)
          m+ (UNLINK-ENV) ) )

(define (CALL1 address m1)
  (append m1 (INVOKE1 address)) )

(define (CALL2 address m1 m2)
  (append m1 (PUSH-VALUE)
          m2 (POP-ARG1)
          (INVOKE2 address) ) )

(define (CALL3 address m1 m2 m3)
  (append m1 (PUSH-VALUE)
          m2 (PUSH-VALUE)
          m3 (POP-ARG2) (POP-ARG1)
          (INVOKE3 address) ) )

(define (FIX-CLOSURE m+ arity)
  (define the-function
    (append (ARITY=? (+ arity 1))
            (EXTEND-ENV) m+
            (RETURN) ) )
  (append (CREATE-CLOSURE 1)
          (GOTO (length the-function))
          the-function ) )

(define (TR-REGULAR-CALL m m*)
  (append m (PUSH-VALUE)
          m* (POP-FUNCTION)
          (FUNCTION-INVOKE) ) )

(define (STORE-ARGUMENT m m* index)
  (append m (PUSH-VALUE)
          m* (POP-FRAME! index) ) )

(define (CONS-ARGUMENT m m* arity)
  (append m (PUSH-VALUE)
          m* (POP-CONS-FRAME! arity) ) )
```

Вызовы `append`, коими кишат наши определения, довольно ощутимо замедляют кодогенерацию. Хорошим решением будет собирать машинные инструкции в последовательность за один проход. Именно так мы и поступим при разработке компилятора в Си. [**см. стр. 449**] Для реализации подобного подхода потребуется вдобавок к линеаризации самого машинного кода также линеаризовать и его генераторы. Например, `CALL2` должна работать в такой последовательности:

1) сгенерировать и вывести код `m1`;
2) вывести код `(PUSH-VALUE)`;
3) сгенерировать и вывести код `m2`;
4) вывести код `(POP-ARG1)`;
5) вывести код `(INVOKE2` *адрес*`)`.

Подобные изменения легко сделать всюду, кроме `GOTO` и `JUMP-FALSE`, так как теперь не получится заранее вычислить смещения при генерации кода для `ALTERNATIVE`, `FIX-CLOSURE` и `NARY-CLOSURE`. Это затруднение можно решить правкой смещений задним числом (такой приём называется *backpatching*). Например, для `FIX-CLOSURE` следует делать так:

1) вывести инструкцию `CREATE-CLOSURE`;
2) вывести инструкцию `GOTO`, но без смещения, запомнив при этом текущее состояние счётчика команд;
3) сгенерировать и вывести код тела функции;
4) зная текущее значение счётчика команд и помня сохранённое ранее, вычислить величину смещения для `GOTO`;
5) записать это число на зарезервированное[1] внутри `GOTO` место.

В результате получится более аккуратная и эффективная процедура генерации кода, чем есть сейчас, но пока мы всё же оставим текущий вариант из-за его простоты.

[1] Вот здесь могут возникнуть некоторые проблемы, потому что смещение может оказаться как меньшим 256, так и превышать это значение, то есть занимать один или два байта. Это может оказаться причиной определённых потерь производительности.

7.2.1. Дизассемблирование

В начале этой главы [**см. стр. 269**] была показана промежуточная форма следующей небольшой программки:

```
((lambda (fact) (fact 5 fact (lambda (x) x)))
 (lambda (n f k) (if (= n 0) (k 1)
                     (f (- n 1) f (lambda (r) (k (* n r)))) )) )
```

Теперь, наконец, мы сможем увидеть, как она выглядит на настоящем машинном языке. Сложно не заметить сходство таблицы 7.3 с ассемблерным листингом.

7.3. Кодирование инструкций

Почти на каждой странице этой главы упоминаются байты, но до сих пор мы не видели ни одного из них живьём. К счастью, чтобы они появились, достаточно лишь по-другому понимать инструкции таблицы 7.2: теперь это не непосредственно исполнимые команды, а генераторы последовательностей байтов — исходного кода для другой, более быстрой функции `run`. Сейчас мы сконцентрируемся именно на этом аспекте байт-кода: его компактности и скорости интерпретации.

Байт-кодирование требует особой осторожности: за каждой инструкцией должно быть закреплено уникальное число, по которому определяется её поведение внутри `run`, длина операндов этой инструкции, а также многое другое. Поэтому, чтобы в процессе ничего не напутать, инструкции будут определяться с помощью макроса `define-instruction`. При этом предполагается, что все определения машинных инструкций на самом деле обёрнуты в макрос `define-instruction-set`, который кроме собственно формы `define-instruction` определяет также несколько вспомогательных функций.

```
(define-syntax define-instruction-set
  (syntax-rules (define-instruction)
    ((define-instruction-set
       (define-instruction (name . args) n . body) ... )
     (begin
       (define (run)
         (let ((instruction (fetch-byte)))
           (case instruction
             ((n) (run-clause args body)) ... ) ) )
         (run) )

       (define (instruction-size code pc)
         (let ((instruction (vector-ref code pc)))
           (case instruction
             ((n) (size-clause args)) ... ) ) )
```

```
(CREATE-CLOSURE 2)
(GOTO 82)
(ARITY=? 4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 0)
(POP-ARG1)
(INVOKE2 #<=>)
(JUMP-FALSE 13)
(SHALLOW-ARGUMENT-REF 2)
(PUSH-VALUE)
(CONSTANT 1)
(PUSH-VALUE)
(ALLOCATE-FRAME 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-INVOKE)
(GOTO 54)
(SHALLOW-ARGUMENT-REF 1)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 1)
(POP-ARG1)
(INVOKE2 #<->)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 1)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(GOTO 24)
(ARITY=? 2)
(EXTEND-ENV)
(DEEP-ARGUMENT-REF 1 2)
(PUSH-VALUE)
(DEEP-ARGUMENT-REF 1 0)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(POP-ARG1)
(INVOKE2 #<*>)
(PUSH-VALUE)
(ALLOCATE-FRAME 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-INVOKE)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 3)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-INVOKE)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 1)
(POP-FRAME! 0)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 5)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(GOTO 6)
(ARITY=? 2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 3)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-INVOKE)
(RETURN)
```

Таблица 7.3. Результат компиляции.

```
        (define (instruction-decode code pc)
          (define (fetch-byte)
            (let ((byte (vector-ref code pc)))
              (set! pc (+ pc 1))
              byte ) )
          (let-syntax
              ((decode-clause
                (syntax-rules ()
                  ((decode-clause iname ())
                   '(iname) )
                  ((decode-clause iname (a))
                   (let ((a (fetch-byte)))
                     (list 'iname a) ) )
                  ((decode-clause iname (a b))
                   (let* ((a (fetch-byte))
                          (b (fetch-byte)) )
                     (list 'iname a b) ) ) ) ))
            (let ((instruction (fetch-byte)))
              (case instruction
                ((n) (decode-clause name args)) ... ) ) ) ) ) ) ) )

(define-syntax run-clause
  (syntax-rules ()
    ((run-clause () body)
     (begin . body) )
    ((run-clause (a) body)
     (let ((a (fetch-byte)))
       . body ) )
    ((run-clause (a b) body)
     (let* ((a (fetch-byte))
            (b (fetch-byte)) )
       . body ) ) ) )

(define-syntax size-clause
  (syntax-rules ()
    ((size-clause ())    1)
    ((size-clause (a))   2)
    ((size-clause (a b)) 3) ) )
```

Макрос `define-instruction` одновременно определяет сразу три функции: функцию `run`, исполняющую байт-коды; функцию `instruction-size`, возвращающую размер инструкций; и функцию `instruction-decode`, переводящую байт-код обратно в символьную форму. Очевидно, `instruction-decode`, будет весьма полезной при отладке. Рассмотрим пример:

```
(define-instruction (SHALLOW-ARGUMENT-REF j) 5
  (set! *val* (activation-frame-argument *env* j)) )
```

Эта форма добавляет в определение `run` обработку инструкции, кодируемой числом 5:

```
(define (run)
  (let ((instruction (fetch-byte)))
    (case instruction
      ...
      ((5) (let ((j (fetch-byte)))
             (set! *val* (activation-frame-argument *env* j)) ))
      ... ) )
  (run) )
```

Функция `fetch-byte` используется для извлечения операндов инструкции из потока байтов. Она очень просто определяется и имеет один побочный эффект: сдвиг `*pc*` на одну позицию вперёд.

```
(define (fetch-byte)
  (let ((byte (vector-ref *code* *pc*)))
    (set! *pc* (+ 1 *pc*))
    byte ) )
```

Также `fetch-byte` используется функцией `run` для считывания следующей инструкции. Кстати, обратите внимание, что счётчик команд всегда указывает на *следующую* инструкцию, а не на текущую исполняемую. Точно такое же решение применяется и во многих настоящих процессорах.

Для достижения действительно высокой скорости интерпретации (другими словами, чтобы получить быструю `run`) следует уделить особое внимание используемой здесь форме `case`. Благодаря тому, что дискриминант `case` может быть только числом от 0 до 255, становится возможным реализовать `case` через таблицу переходов, что позволит выполнять выбор нужной ветви за постоянное время. Если же `case` раскрывается в цепочку `(if (eqv? ...) ...)`, то выбор выполняется за линейное время — прощай, производительность! Лишь некоторые из компиляторов Лиспа умеют выполнять такую оптимизацию автоматически (среди них Sqil [Sén91] и Bigloo [Ser93]).

`define-instruction` также принимает участие в определении функции `instruction-size`: туда она добавляет информацию о том, что инструкция `SHALLOW-ARGUMENT-REF` занимает два байта. Аналогично, здесь `case` можно заменить вектором, хранящим те же самые числа.

```
(define (instruction-size code pc)
  (let ((instruction (vector-ref code pc)))
    (case instruction
      ...
      ((5) 2)
      ... ) ) )
```

Наконец, `define-instruction` добавляет в `instruction-decode`, ветку для обработки инструкции `SHALLOW-ARGUMENT-REF`. Функция `instruction-decode`

использует свою личную реализацию `fetch-byte`, чтобы вести себя подобно `run`, но не влиять на глобальный счётчик команд.

```
(define (instruction-decode code pc)
  (define (fetch-byte)
    (let ((byte (vector-ref code pc)))
      (set! pc (+ pc 1))
      byte ) )
  (let ((instruction (vector-ref code pc)))
    (case instruction
      ...
      ((5) (let ((j (fetch-byte)))
             (list 'SHALLOW-ARGUMENT-REF j) ))
      ... ) ) )
```

7.4. Инструкции

Всего существует 256 возможных кодов инструкций, мы же используем только 34 из них. Настало время задействовать эти резервы; вспомните, как мы предлагали объединять инструкции, которые часто используются вместе. Кроме того очевиден [Cha80] тот факт, что большинство функций имеет небольшое число аргументов. Действительно, достаточно лишь посмотреть на функции, использованные в данной книге. Их анализ показывает следующие результаты: лишь 16 функций имеют переменную арность; распределение арности остальных 1988 приведено в таблице 7.4.

арность	0	1	2	3	4	5	6	7	8
доля, %	35	30	18	9	4	1	0	0	0
накопление, %	35	66	84	93	97	99	99	99	100

Таблица 7.4. Распределение функций по количеству аргументов.

Как видим, подавляющее большинство функций имеет не более четырёх аргументов. При обобщении этих результатов следует учесть, что настолько большие числа для нулевой арности получены в основном благодаря шестой главе. Итого, однозначно имеет смысл оптимизировать вызовы функций, принимающих менее четырёх аргументов. Этим мы и займёмся, используя избыточные коды инструкций для создания специализированных вариантов.

7.4.1. Локальные переменные

Пожалуй, давайте начнём с `SHALLOW-ARGUMENT-REF`. Эта инструкция имеет один операнд: j, а её действие заключается в том, что регистр `*val*` по-

лучает значение j-й локальной переменной текущей записи активации, хранящейся в регистре `*env*`. Здесь можно ввести специализации для случаев $j = 0, 1, 2, 3$, сделав каждый из них отдельной инструкцией со своим кодом. Также для простоты можно ввести ограничение[2] в 256 аргументов функции — это позволит всегда кодировать индекс аргумента одним-единственным байтом. Функция `check-byte`[3] убеждается в соблюдении этого ограничения. Итак, вот определение `SHALLOW-ARGUMENT-REF`. Эта инструкция является генератором байт-кодов — она возвращает список[4] байтов, соответствующих кодируемой инструкции. Этот список состоит из одного или двух байтов, в зависимости от операнда.

```
(define (SHALLOW-ARGUMENT-REF j)
  (check-byte j)
  (case j
    ((0 1 2 3) (list (+ 1 j)))
    (else      (list 5 j)) ) )

(define (check-byte j)
  (unless (<= 0 j 255)
    (static-wrong "Cannot pack a number within a byte" j) ) )
```

А вот определения собственно пяти[5] специализаций:

```
(define-instruction (SHALLOW-ARGUMENT-REF0) 1
  (set! *val* (activation-frame-argument *env* 0)) )
(define-instruction (SHALLOW-ARGUMENT-REF1) 2
  (set! *val* (activation-frame-argument *env* 1)) )
(define-instruction (SHALLOW-ARGUMENT-REF2) 3
  (set! *val* (activation-frame-argument *env* 2)) )
(define-instruction (SHALLOW-ARGUMENT-REF3) 4
  (set! *val* (activation-frame-argument *env* 3)) )
(define-instruction (SHALLOW-ARGUMENT-REF j) 5
  (set! *val* (activation-frame-argument *env* j)) )
```

`SET-SHALLOW-ARGUMENT!` тоже работает с локальными переменными, так что определяется аналогично. (Пропущенные здесь специализации легко определить самостоятельно.)

---

[2] В Common Lisp даже есть специальная константа `LAMBDA-PARAMETERS-LIMIT`, хранящая максимально возможно количество аргументов функции. По стандарту оно не может быть меньше 50. Стандарт Scheme ничего не говорит о подобных ограничениях, но его молчание можно толковать по-разному.

Как бы то ни было, вопрос представления чисел интересен сам по себе. В большинстве систем, поддерживающих длинную арифметику, максимально представимое число немного меньше, чем $256^{2^{32}}$. (Чуть-чуть не хватило до бесконечности.) Хитрость в том, что перед длинным числом записывается его длина в байтах. Аналогичное решение можно применить и здесь.

[3] В последующих определениях мы не будем упоминать `check-byte` ради краткости.

[4] Да-да, `list`, `append`, неэффективно, страница 281. Извините.

[5] Инструкции с кодом `0` не существует — бедный ноль и без того особенный.

```
(define (SET-SHALLOW-ARGUMENT! j)
  (case j
    ((0 1 2 3) (list (+ 21 j)))
    (else      (list 25 j)) ) )

(define-instruction (SET-SHALLOW-ARGUMENT!2) 23
  (set-activation-frame-argument! *env* 2 *val*) )
(define-instruction (SET-SHALLOW-ARGUMENT! j) 25
  (set-activation-frame-argument! *env* j *val*) )
```

Что касается свободных переменных, то мы предполагаем,[6] что их индексы равновероятны, поэтому для них специализаций не будет:

```
(define (DEEP-ARGUMENT-REF i j) (list 6 i j))
(define (SET-DEEP-ARGUMENT! i j) (list 26 i j))

(define-instruction (DEEP-ARGUMENT-REF i j) 6
  (set! *val* (deep-fetch *env* i j)) )
(define-instruction (SET-DEEP-ARGUMENT! i j) 26
  (deep-update! *env* i j *val*) )
```

7.4.2. Глобальные переменные

Про распределение глобальных изменяемых переменных, определяемых пользователем, виртуальная машина ничего знать не может в принципе, так что они тоже будут кодироваться без специализаций. Для простоты предположим, что глобальных переменных может быть не более 256, что позволит закодировать их адрес одним байтом. Таким образом, имеем следующее:

```
(define (GLOBAL-REF i) (list 7 i))
(define (CHECKED-GLOBAL-REF i) (list 8 i))
(define (SET-GLOBAL! i) (list 27 i))

(define-instruction (GLOBAL-REF i) 7
  (set! *val* (global-fetch i)) )

(define-instruction (CHECKED-GLOBAL-REF i) 8
  (set! *val* (global-fetch i))
  (when (eq? *val* undefined-value)
    (signal-exception #t (list "Uninitialized global variable" i)) ) )

(define-instruction (SET-GLOBAL! i) 27
  (global-update! i *val*) )
```

[6] Вообще-то это не совсем верное предположение, ведь мы только что убедились в том, что j чаще всего оказывается меньше четырёх. Но так как свободные переменные используются значительно реже локальных и глобальных, то можно не особо спешить с ускорением доступа к ним.

Случай предопределённых неизменяемых переменных более интересен, так как доступ к наиболее часто используемым из них можно немного ускорить. В список таких переменных включены `t`, `f`, `nil`, `cons`, `car` и некоторые другие.

```
(define (PREDEFINED i)
  (check-byte i)
  (case i
    ;; 0 = #t, 1 = #f, 2 = (), 3 = cons, 4 = car,
    ;; 5 = cdr, 6 = pair?, 7 = symbol?, 8 = eq?
    ((0 1 2 3 4 5 6 7 8) (list (+ 10 i)))
    (else                (list 19 i)) ) )

(define-instruction (PREDEFINED0) 10    ; #t
  (set! *val* #t) )
(define-instruction (PREDEFINED i) 19
  (set! *val* (predefined-fetch i)) )
```

Раз уж некоторые константы трактуются по-особому, то давайте так же поступим и с цитатами. Предположим, в машине имеется регистр `*constants*`, содержащий вектор всех используемых в программе цитат, тогда доступ к ним можно получить с помощью функции `quotation-fetch`:

```
(define (quotation-fetch i)
  (vector-ref *constants* i) )
```

Комбинатор `CONSTANT` собирает все цитаты в переменной `*quotations*` во время компиляции. Перед исполнением программы все используемые цитаты загружаются в регистр `*constants*`. Цитаты не равновероятны: некоторые из них используются чаще других, поэтому им будут выданы собственные инструкции. Часть из них по очевидным причинам совпадает с предопределёнными константами вроде `PREDEFINED0`. Наконец, мы предположим, что цитировать напрямую в машинном коде можно только целые числа от 0 до 255, остальные будут на общих правах размещаться в `*constants*`.[7]

```
(define (CONSTANT value)
  (cond ((eq? value #t)    (list 10))
        ((eq? value #f)    (list 11))
        ((eq? value '())   (list 12))
        ((equal? value -1) (list 80))
        ((equal? value 0)  (list 81))
        ((equal? value 1)  (list 82))
        ((equal? value 2)  (list 83))
        ((equal? value 3)  (list 84))
        ((and (integer? value)   ; непосредственное значение
              (<= 0 value 255) )
         (list 79 value) )
        (else (EXPLICIT-CONSTANT value)) ) )
```

[7] Это досадно, но где-то же должна проходить граница между обычными числами и длинными.

```
(define (EXPLICIT-CONSTANT value)
  (set! *quotations* (append *quotations* (list value)))
  (list 9 (- (length *quotations*) 1)) )

(define-instruction (CONSTANT-1) 80
  (set! *val* -1) )
(define-instruction (CONSTANT0) 81
  (set! *val* 0) )
(define-instruction (SHORT-NUMBER value) 79
  (set! *val* value) )
```

7.4.3. Переходы

Если вы думаете, что расставленные то тут, то там ограничения чересчур суровы, тогда этот раздел для вас. Очевидно, что не стоит ограничивать `GOTO` или `JUMP-FORWARD` прыжками максимум на 256 байтов вперёд,[8] так как код без проблем может превысить эти размеры. Поэтому мы будем различать случаи, когда длина прыжка записывается одним или двумя байтами. Введём две отдельные инструкции[9]: `SHORT-GOTO` и `LONG-GOTO`. Думаем, этого будет достаточно на время; наш компилятор пока не метит в олимпийские чемпионы, чтобы прыгать дальше 64 килобайт.

```
(define (GOTO offset)
  (cond ((< offset 256) (list 30 offset))
        ((< offset 65536)
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
           (list 28 offset1 offset2) ) )
        (else (static-wrong "Too far jump" offset)) ) )

(define (JUMP-FALSE offset)
  (cond ((< offset 256) (list 31 offset))
        ((< offset 65536)
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
           (list 29 offset1 offset2) ) )
        (else (static-wrong "Too far jump" offset)) ) )

(define-instruction (SHORT-GOTO offset) 30
  (set! *pc* (+ *pc* offset)) )
```

[8] На некоторых старых машинах, основанных на процессоре i8086, такое ограничение действительно существовало.

[9] Очевидно, что подобное разделение на `SHORT`- и `LONG`-версию может быть применено, например, к `GLOBAL-REF` с компанией, чтобы количество изменяемых глобальных переменных больше не было ограничено 256-ю.

```
(define-instruction (SHORT-JUMP-FALSE offset) 31
  (if (not *val*) (set! *pc* (+ *pc* offset))) )

(define-instruction (LONG-GOTO offset1 offset2) 28
  (let ((offset (+ offset1 (* 256 offset2))))
    (set! *pc* (+ *pc* offset)) ) )
```

7.4.4. Вызовы функций

Для начала разберёмся с обычными вызовами, а потом перейдём ко встраиваемым. Наблюдение о преимущественно небольшой арности используемых функций будет полезным и здесь. Выделив ещё немного байт-кодов, мы можем специализировать создание записей активаций для малого числа аргументов:

```
(define (ALLOCATE-FRAME size)
  (case size
    ((0 1 2 3 4) (list (+ 50 size)))
    (else        (list 55 (+ size 1))) ) )

(define-instruction (ALLOCATE-FRAME1) 50
  (set! *val* (allocate-activation-frame 1)) )
(define-instruction (ALLOCATE-FRAME size+1) 55
  (set! *val* (allocate-activation-frame size+1)) )
```

Раскладывание значений аргументов по записям тоже можно улучшить для небольших арностей:

```
(define (POP-FRAME! index)
  (case index
    ((0 1 2 3) (list (+ 60 index)))
    (else      (list 64 index)) ) )

(define-instruction (POP-FRAME!0) 60
  (set-activation-frame-argument! *val* 0 (stack-pop)) )
(define-instruction (POP-FRAME! index) 64
  (set-activation-frame-argument! *val* index (stack-pop)) )
```

Встраиваемые вызовы встречаются очень часто, поэтому им стоит выдать личные коды. Например, комбинатор `INVOKE1` отвечает за унарные предопределённые функции:

```
(define (INVOKE1 address)
  (case address
    ((car)     (list 90))
    ((cdr)     (list 91))
    ((pair?)   (list 92))
    ((symbol?) (list 93))
```

```
      ((display) (list 94))
      (else
       (static-wrong "Cannot inline" address) ) ) )

(define-instruction (CALL1-car) 90
  (set! *val* (car *val*)) )
(define-instruction (CALL1-cdr) 91
  (set! *val* (cdr *val*)) )
```

Естественно, аналогичные действия выполняются также для арностей 0, 2 и 3. Функция `display` сделана встраиваемой исключительно ради облегчения отладки. Вообще ей там не место, учитывая её сложность и скорость работы. Лучше потратить байт-коды на что-то более полезное, например: `cadr`, `cddr`, `cdddr` (в порядке убывания полезности).

Конечно, и сами проверки арности тоже можно специализировать для наиболее часто встречающихся случаев:

```
(define (ARITY=? arity+1)
  (case arity+1
    ((1 2 3 4) (list (+ 70 arity+1)))
    (else      (list 75 arity+1)) ) )

(define-instruction (ARITY=?2) 72
  (unless (= (activation-frame-argument-length *val*) 2)
    (signal-exception
     #f (list "Incorrect arity for unary function") ) ) )

(define-instruction (ARITY=? arity+1) 75
  (unless (= (activation-frame-argument-length *val*) arity+1)
    (signal-exception #f (list "Incorrect arity") ) ) )
```

Учитывая весьма небольшую долю функций с переменной арностью, для них особого выигрыша подобные специализации не дадут.

Сейчас встраиваются лишь простые и быстрые функции. Если у нас останутся свободные байты, то можно начать встраивать и функции посложнее, вроде `memq` или `equal?`. Единственное возможное неудобство заключается в том, что вызов `memq` может никогда не завершиться (или же просто будет выполняться очень долго), если просматриваемый ей список имеет циклы. MacScheme [85M85], к примеру, ограничивает для `memq` максимальную просматриваемую длину списка несколькими тысячами элементов.

7.4.5. Прочее

Ещё у нас остаются несколько простых инструкций, не имеющих операндов и кодируемых одним-единственным байтом. Генераторы байт-кодов для них одинаковы, так что хватит и одного примера:

```
(define (RESTORE-ENV) (list 38))
```

А вот и их определения:

```
(define-instruction (EXTEND-ENV) 32
  (set! *env* (sr-extend* *env* *val*)) )

(define-instruction (UNLINK-ENV) 33
  (set! *env* (activation-frame-next *env*)) )

(define-instruction (PUSH-VALUE) 34
  (stack-push *val*) )

(define-instruction (POP-ARG1) 35
  (set! *arg1* (stack-pop)) )

(define-instruction (POP-ARG2) 36
  (set! *arg2* (stack-pop)) )

(define-instruction (CREATE-CLOSURE offset) 40
  (set! *val* (make-closure (+ *pc* offset) *env*)) )

(define-instruction (RETURN) 43
  (set! *pc* (stack-pop)) )

(define-instruction (FUNCTION-GOTO) 46
  (invoke *fun* #t) )

(define-instruction (FUNCTION-INVOKE) 45
  (invoke *fun* #f) )

(define-instruction (POP-FUNCTION) 39
  (set! *fun* (stack-pop)) )

(define-instruction (PRESERVE-ENV) 37
  (preserve-environment) )

(define-instruction (RESTORE-ENV) 38
  (restore-environment) )
```

Сохранение и восстановление окружения реализуется следующей парой функций:

```
(define (preserve-environment)
  (stack-push *env*) )

(define (restore-environment)
  (set! *env* (stack-pop)) )
```

Возможны два способа вызова функций: `FUNCTION-GOTO` для выполнения вызовов из хвостовой позиции и `FUNCTION-INVOKE` для всего остального. При

вызове из хвостовой позиции функция просто использует тот адрес возврата, который в момент её вызова лежал на верхушке стека. При нормальном вызове поток исполнения необходимо вернуть к инструкции, следующей за `(FUNCTION-INVOKE)`, поэтому она должна самостоятельно сохранить состояние счётчика команд в стеке. Хвостовой вызов же по определению эквивалентен `FUNCTION-INVOKE`, за которой сразу же следует `RETURN`, а так любая функция всегда завершается инструкцией `RETURN`, то достаточно просто оставить в стеке старый адрес возврата, чтобы `RETURN` вызываемой функции перешла куда следует — ведь вызывающей функции всё равно ничего больше делать не надо. Именно так и поступает `FUNCTION-GOTO`. Обобщённая функция `invoke` реализует оба варианта поведения.

```
(define-generic (invoke (f) tail?)
  (signal-exception #f (list "Not a function" f)) )

(define-method (invoke (f closure) tail?)
  (unless tail? (stack-push *pc*))
  (set! *env* (closure-closed-environment f))
  (set! *pc* (closure-code f)) )
```

Так как `invoke` является обобщённой, это позволяет распространить её и на другие объекты: например, на примитивы, представляемые замыканиями.

```
(define-method (invoke (f primitive) tail?)
  (unless tail? (stack-push *pc*))
  ((primitive-address f)) )
```

Естественно, значение `tail?` надо учитывать при любом вызове функции. Для примитивов тоже важно, прерываются ли данным вызовом вычисления или же наоборот, сводятся к нему.

7.5. Запускаем компилятор-интерпретатор

Для реализации REPL (что мы делали во всех предыдущих главах) потребуется слаженное взаимодействие компилятора и виртуальной машины. Функция `standalone-producer-c7` принимает программу и инициализирует компилятор. Под инициализацией здесь понимается создание начального глобального окружения и перечня используемых цитат. Результатом функции `meaning` теперь является список байтов, который укладывается в вектор между инструкциями `FINISH` в начале и `RETURN` в конце. Начальным состоянием счётчика программ будет единица — адрес первой инструкции после пролога. По известным причинам [**см. упр. 6.1**], с которыми мы столкнёмся позже, эта функция также инициализирует поимённый список глобальных переменных и цитат. В конце концов мы получаем замыкание, представляющее наши вычисления и ожидающее последний важный параметр — необходимый размер стека.

```
(define (chapter7-interpreter)
  (define (toplevel)
    (display ((standalone-producer-c7 (read)) 100))
    (toplevel) )
  (toplevel) )

(define (standalone-producer-c7 e)
  (set! g.current (original.g.current))
  (set! *quotations* '())
  (let* ((code (make-code-segment (meaning e r.init #t)))
         (start-pc (length (code-prologue)))
         (global-names (map car (reverse g.current)))
         (constants (apply vector *quotations*)) )
    (lambda (stack-size)
      (run-machine stack-size start-pc code
                   constants global-names ) ) ) )

(define (make-code-segment m)
  (apply vector (append (code-prologue) m (RETURN))) )

(define (code-prologue)
  (set! finish-pc 0)
  (FINISH) )
```

Функция `run-machine` инициализирует состояние виртуальной машины и запускает её. Она должна создать вектор глобальных изменяемых переменных, запомнить их имена, создать стек и инициализировать регистры. Вроде бы ничего не забыли… стоп! а как нам остановить машину? Откомпилированная программа представляется в виде огромной функции, которую виртуальная машина вызывает из хвостовой позиции, то есть последней выполняемой инструкцией программы будет `RETURN`. Чтобы всё отработало как надо, в стеке должен лежать адрес возврата для этой инструкции. Теперь понятно, зачем в самом начале вектора кода располагается инструкция `FINISH`? Сама она определяется так:

```
(define-instruction (FINISH) 20
  (*exit* *val*) )
```

Её задачей будет остановить виртуальную машину и передать наверх значение регистра `*val*` на момент завершения работы. Такого поведения мы добиваемся с помощью ловкой активации продолжения функции `run-machine`.

```
(define (run-machine stack-size pc code constants global-names)
  (set! sg.current (make-vector (length global-names) undefined-value))
  (set! sg.current.names global-names)
  (set! *constants*   constants)
  (set! *code*        code)
  (set! *env*         sr.init)
```

```
(set! *stack*       (make-vector stack-size))
(set! *stack-index* 0)
(set! *val*         'anything)
(set! *fun*         'anything)
(set! *arg1*        'anything)
(set! *arg2*        'anything)
(stack-push finish-pc)          ; точка выхода из программы
(set! *pc*          pc)
(call/cc (lambda (exit)
           (set! *exit* exit)
           (run) )) )
```

7.5.1. Передышка

Напоследок, после тщательного рассмотрения инструкций нашей виртуальной машины, давайте ещё раз взглянем на результат компиляции примера с факториалом, теперь уже под новый диалект машинного языка. И оформленный, естественно, не в виде байтов, а как читабельный дизассемблерный листинг. Прошу любить и жаловать: таблица 7.5.

7.6. Продолжения

Мы уже знаем, что `call/cc` — это магический оператор, реифицирующий контекст исполнения программы, превращающий его в полноценный объект языка, который можно сохранить в переменной, а потом восстановить обратно. Настало время объяснить, как именно он работает. Предлагаемая реализация является канонической. Более эффективные, но и более сложные варианты можно найти в [CHO88, HDB90, MB93].

Контекст исполнения программы — это стек и ничего, кроме стека. Действительно, регистры `*fun*`, `*arg1*` и `*arg2*` используются лишь для хранения промежуточных результатов вычислений, их значения лишены смысла между выражениями компилируемого языка. (Нельзя[10] вызвать `call/cc` или что-нибудь ещё во время процесса обработки вызова функции, когда эти регистры используются.)

Регистр `*val*` используется для передачи значений самим продолжениям, так что тут нечего сохранять. Регистр `*env*` тоже бессмысленно запоминать, потому как `call/cc` — это не встраиваемый примитив; если окружение надо было сохранить, то это уже сделано в силу самого вызова *функции* `call/cc`. Таким образом, остаётся сохранить только стек. Следующие функции помогут работать с ним:

[10] Говоря в общем, это всё же возможно при обработке асинхронных вызовов вроде сигналов Un*x. В таком случае, очевидно, следует использовать время от времени проверяемый флаг, как в [Dev85], или иной механизм синхронизации.

```
(CREATE-CLOSURE 2)
(SHORT-GOTO 60)
(ARITY=?4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT0)
(POP-ARG1)
(CALL2-=)
(SHORT-JUMP-FALSE 10)
(SHALLOW-ARGUMENT-REF2)
(PUSH-VALUE)
(CONSTANT1)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(SHORT-GOTO 40)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CONSTANT1)
(POP-ARG1)
(CALL2--)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF1)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 19)
(ARITY=?2)
(EXTEND-ENV)
(DEEP-ARGUMENT-REF 1 2)
(PUSH-VALUE)
(DEEP-ARGUMENT-REF 1 0)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(POP-ARG1)
(CALL2-*)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME2)
(POP-FRAME!0)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(SHORT-NUMBER 5)
(PUSH-VALUE)
(SHALLOW-ARGUMENT-REF0)
(PUSH-VALUE)
(CREATE-CLOSURE 2)
(SHORT-GOTO 4)
(ARITY=?2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME4)
(POP-FRAME!2)
(POP-FRAME!1)
(POP-FRAME!0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
```

Таблица 7.5. Откомпилированная программа.

```
(define (save-stack)
  (let ((copy (make-vector *stack-index*)))
    (vector-copy! *stack* copy 0 *stack-index*)
    copy ) )

(define (restore-stack copy)
  (set! *stack-index* (vector-length copy))
  (vector-copy! copy *stack* 0 *stack-index*) )

(define (vector-copy! old new start end)
  (let copy ((i start))
    (when (< i end)
          (vector-set! new i (vector-ref old i))
          (copy (+ i 1)) ) ) )
```

Продолжения имеют личный класс значений и особый протокол вызова. При активации продолжения следует восстановить стек, положить переданное значение в регистр `*val*`, а затем выполнить `RETURN`, который перейдёт к инструкции, следующей за вызовом `call/cc`. Очевидно, для этого `call/cc` должна вызываться исключительно с помощью `FUNCTION-INVOKE`, а значит, следующей исполняемой инструкцией обязательно будет `RESTORE-ENV`. (Можете убедиться в этом, перечитав определение `REGULAR-CALL`.) [**см. стр. 276**]

```
(define-class continuation Object
  ( stack ) )

(define-method (invoke (f continuation) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (begin
        (restore-stack (continuation-stack f))
        (set! *val* (activation-frame-argument *val* 0))
        (set! *pc* (stack-pop)) )
      (signal-exception #f (list "Incorrect arity" 'continuation)) ) )
```

Для правильной работы продолжений `call/cc` должна организовать стек таким образом, чтобы после активации продолжения начал исполняться код, следующий за самой `call/cc`. То есть форма `(call/cc` f`)` с точки зрения стека должна быть идентична вызову функции f из хвостовой позиции (если забыть про `dynamic-wind`). Следующее определение добивается этого, выполняя хвостовой вызов вручную. Функция `call/cc` сама создаёт нужную запись активации для своего аргумента-замыкания, сохраняет в ней продолжение, помещает эту запись в обход стека сразу же в `*val*` и вызывает f напрямую.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-primitive
      (lambda ()
```

```
(if (= arity+1 (activation-frame-argument-length *val*))
    (let ((f (activation-frame-argument *val* 0))
          (frame (allocate-activation-frame (+ 1 1))) )
      (set-activation-frame-argument!
       frame 0 (make-continuation (save-stack)) )
      (set! *val* frame)
      (set! *fun* f)       ; полезно для отладки
      (invoke f #t) )
    (signal-exception #t (list "Incorrect arity"
                               'call/cc )) ) ) ) ) )
```

Итого, каноническая реализация `call/cc` выполняет одно полное копирование стека при своём вызове, а также по дополнительному копированию при каждой активации сохранённых продолжений. Естественно, существуют стратегии поумнее, но они более сложны для понимания и реализации. Основываются они на том, что у захватываемых продолжений нередко [Dan87] имеются общие части, которые бессмысленно дублировать, поэтому можно оптимизировать `call/cc`, если использовать более разумный способ представления продолжений, нежели полные независимые снимки стека.

7.7. Переходы

Несмотря на то, что стоимость `call/cc` можно уменьшить, обычные переходы (escapes) всё ещё остаются гораздо более эффективными. Мы упоминали об этом в третьей главе, так что сейчас было бы нечестным не показывать, как они реализуются технически. Для иллюстрации была выбрана конструкция `bind-exit`. [**см. стр. 131**] Родом она из диалекта Dylan, её аналогами являются `let/cc` в EuLisp, `block`/`return-from` в Common Lisp, `escape` в VLISP [Cha80].

Наперекор традициям Scheme, она будет реализована в виде специальной формы, а не функции. Во-первых, в данном случае легче реализовать именно специальную форму, так как это избавит нас от излишней работы по соблюдению протокола вызова функций; во-вторых, главной задачей этой главы является демонстрация внутреннего строения компилятора, а не следование догмам. Если вас эти слова не убедили, то вспомните, что подобную специальную форму можно выразить через аналогичную функцию и наоборот, так что это лишь вопрос внешнего вида, а не сути.

Форма `bind-exit` имеет следующий синтаксис:

`(bind-exit (`*переменная*`)` *формы*`...)`

Переменная связывается с продолжением формы `bind-exit`, после чего последовательно вычисляются *формы*. Захваченным продолжением можно воспользоваться только во время этих вычислений. Если у нас есть `bind-exit`, то можно легко определить функцию `call/ep` и наоборот:

```
(define (call/ep f)
  (bind-exit (k) (f k)) )
```

(bind-exit (*k*) *тело*) ≡ (call/ep (lambda (*k*) *тело*))

Переходы представляются объектами класса escape. Его единственное поле хранит высоту стека, какой она была на момент начала вычисления формы bind-exit.

```
(define-class escape Object
  ( stack-index ) )
```

Для определения специальной формы сперва необходимо добавить соответствующую обработку в функцию meaning, синтаксический анализатор компилируемых форм. Её задачей является распознавание наших нововведений:

```
... ((bind-exit) (meaning-bind-exit (caadr e) (cddr e) r tail?)) ...
```

Затем определяется способ предобработки данных форм:

```
(define (meaning-bind-exit n e+ r tail?)
  (let* ((r2 (r-extend* r (list n)))
         (m+ (meaning-sequence e+ r2 #t)) )
    (ESCAPER m+) ) )
```

Обработка заключается в объединении форм, составляющих тело bind-exit, в последовательность, которая вычисляется в лексическом окружении, расширенном локальной переменной, вводимой формой bind-exit. Генерацией байт-кода будет заниматься функция ESCAPER. Для её реализации нам потребуется пара новых инструкций: PUSH-ESCAPER и POP-ESCAPER.

```
(define (ESCAPER m+)
  (append (PUSH-ESCAPER (+ 1 (length m+)))
          m+ (RETURN) (POP-ESCAPER)) )

(define (POP-ESCAPER) (list 250))
(define (PUSH-ESCAPER offset) (list 251 offset))

(define escape-tag (list '*ESCAPE*))

(define-instruction (POP-ESCAPER) 250
  (let* ((tag (stack-pop))
         (escape (stack-pop)) )
    (restore-environment) ) )

(define-instruction (PUSH-ESCAPER offset) 251
  (preserve-environment)
  (let* ((escape (make-escape (+ *stack-index* 3)))
         (frame (allocate-activation-frame 1)) )
    (set-activation-frame-argument! frame 0 escape)
```

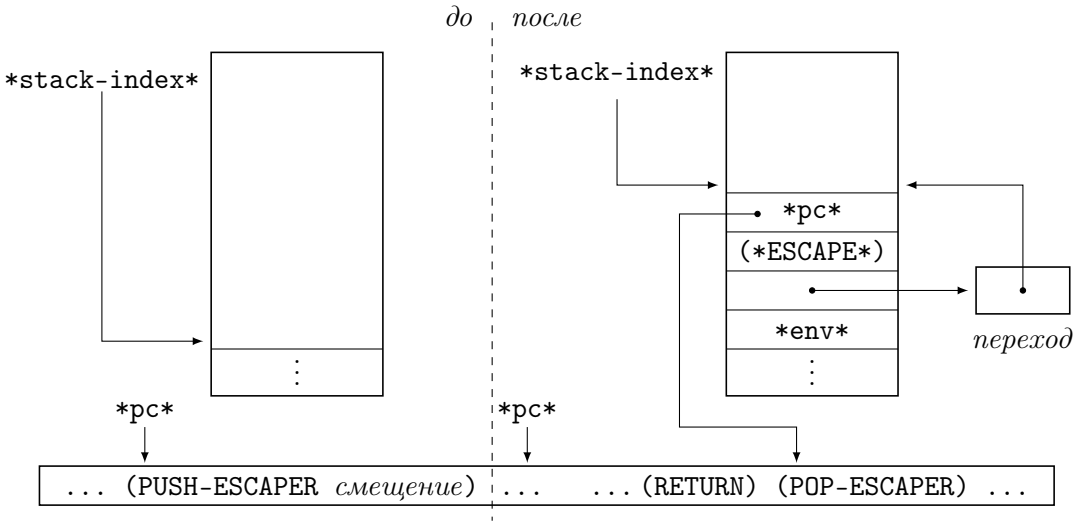


Рис. 7.4. Состояние стека до и после инструкции PUSH-ESCAPER.

```
(set! *env* (sr-extend* *env* frame))
(stack-push escape)
(stack-push escape-tag)
(stack-push (+ *pc* offset)) ) )
```

Иллюстрация принципа работы инструкции PUSH-ESCAPER приведена на рисунке 7.4. Вот что она делает:

1) Сохраняет в стеке текущее окружение.

2) Создаёт дескриптор перехода, указывающий на три ячейки выше текущего положения верхушки стека.

3) Создаёт запись активации с единственным полем, в которое укладывает созданный ранее дескриптор, после чего расширяет текущее окружение полученной записью.

4) Кладёт в стек дескриптор перехода, затем странную метку (*ESCAPE*) и, наконец, адрес инструкции POP-ESCAPER, соответствующей создаваемому переходу.

Процедура выполнения перехода несколько усложняется из-за того, что необходимо проверять, можно ли его вообще выполнить. Переход допустим, если 1) он выполняется вниз по стеку; 2) искомая ячейка стека действительно содержит настоящий дескриптор перехода; 3) это именно тот дескриптор, который нам нужен. Соблюдение всех этих условий обеспечивается предикатом escape-valid?.

```
(define-method (invoke (f escape) tail?)
  (if (= (+ 1 1) (activation-frame-argument-length *val*))
      (if (escape-valid? f)
          (begin (set! *stack-index* (escape-stack-index f))
                 (set! *val* (activation-frame-argument *val* 0))
                 (set! *pc* (stack-pop)) )
          (signal-exception #f (list "Escape out of extent" f)) )
      (signal-exception #f (list "Incorrect arity" 'escape)) ) )

(define (escape-valid? f)
  (let ((index (escape-stack-index f)))
    (and (>= *stack-index* index)
         (eq? f (vector-ref *stack* (- index 3)))
         (eq? escape-tag (vector-ref *stack* (- index 2))) ) ) )
```

При выполнении перехода сначала проверяется арность соответствующей формы, затем стек откатывается до состояния, которое он имел при входе в форму `bind-exit`, после этого аргумент перебрасывается в регистр `*val*` и, наконец, выполняется аналог `RETURN` для выхода из тела формы `bind-exit`. Следующей исполняемой инструкцией будет `POP-ESCAPER`, которая уберёт со стека теперь уже ненужный дескриптор перехода и восстановит окружение.

На случай, если во время вычисления тела формы `bind-exit` переход инициирован не будет, в конце располагается настоящая инструкция `RETURN`, что в итоге приводит к такому же поведению и завершающему состоянию стека, как если бы переход был выполнен.

Вход в форму `bind-exit` вызывает создание двух объектов и занимает четыре ячейки стека. Выполнение перехода стоит одной проверки на допустимость и нескольких операций с регистрами. Дескрипторы переходов создаются и сохраняются явно, так как переменные, создаваемые `bind-exit`, могут быть захвачены замыканиями, что даст им неограниченное время жизни (но не соответствующим переходам). Если бы эти переменные гарантированно не захватывались, то можно было бы ограничиться запоминанием высоты стека в каком-нибудь регистре. Приведённая реализация, признаем, является довольно простой и прямолинейной, поэтому, естественно, и менее эффективной по сравнению с результатами хорошей компиляции. Однако, даже такая реализация оказывается значительно быстрее канонического варианта `call/cc`.

Специальная форма `bind-exit` (вместе с динамическими переменными, которые мы рассмотрим далее) облегчает написание аналогов `catch`/`throw`. [**см. стр. 103**] Также напомним, что если в языке реализована форма `unwind-protect`, то `bind-exit` уже не обязательно гарантирует сверхбыстрые переходы, так как во время их выполнения необходимо выполнять и установленные `unwind-protect` формы-уборщики. Это значит, что нам придётся полностью просматривать часть стека, которую текущая реализация просто игнорирует.

7.8. Динамические переменные

Мы уже не раз упоминали значимость идеи динамических переменных для языков программирования. Проще всего они реализуются с помощью дальнего связывания. Как и ранее, [**см. стр. 207**], мы введём две специальные формы для создания и использования подобных переменных. Вот их синтаксис:

```
(dynamic-let (переменная значение) тело...)
(dynamic переменная)
```

Форма `dynamic-let` связывает *переменную* со *значением* на время вычисления *тела*. Текущее значение динамической переменной можно получить с помощью формы `dynamic`. Естественно, если переменная не была ранее связана с каким-либо значением, то возникает ошибка.

Итак, добавляем две новые формы в синтаксический анализатор `meaning`:

```
...
((dynamic)     (meaning-dynamic-reference (cadr e) r tail?))
((dynamic-let) (meaning-dynamic-let (car (cadr e))
                                    (cadr (cadr e))
                                    (cddr e) r tail? ))
...
```

Реализуем их обработку:

```
(define (meaning-dynamic-let n e e+ r tail?)
  (let ((index (get-dynamic-variable-index n))
        (m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (DYNAMIC-PUSH index) m+ (DYNAMIC-POP)) ) )

(define (meaning-dynamic-reference n r tail?)
  (let ((index (get-dynamic-variable-index n)))
    (DYNAMIC-REF index) ) )
```

В результате получаем три новых генератора:

```
(define (DYNAMIC-PUSH index) (list 242 index))
(define (DYNAMIC-POP)        (list 241))
(define (DYNAMIC-REF index)  (list 240 index))
```

Эти генераторы соответствуют трём новым инструкциям виртуальной машины:

```
(define-instruction (DYNAMIC-PUSH index) 242
  (push-dynamic-binding index *val*) )

(define-instruction (DYNAMIC-POP) 241
  (pop-dynamic-binding) )
```

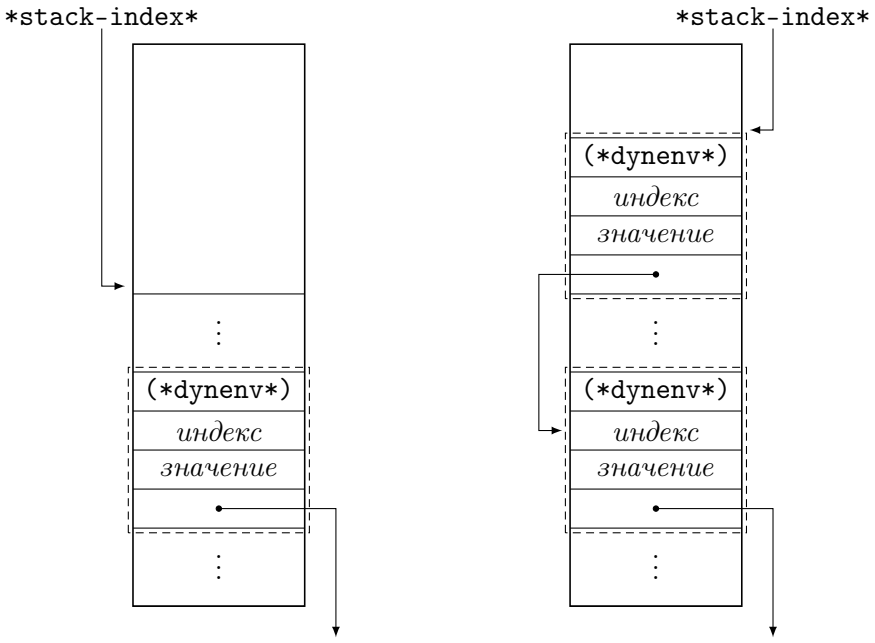


Рис. 7.5. Состояние стека до и после инструкции DYNAMIC-PUSH.

```
(define-instruction (DYNAMIC-REF index) 240
  (set! *val* (find-dynamic-value index)) )
```

Итак, как же нам представить окружение динамических переменных? Первая же идея, которая приходит в голову, — это ввести новый регистр, скажем, *dynenv*, хранящий ассоциативный список имён и значений динамических переменных. Вернее, цепочку *фреймов*, подобную стеку записей активаций. К сожалению, в таком случае этот регистр будет являться частью состояния машины, за сохранение/восстановление которого отвечают инструкции PRESERVE-ENV и RESTORE-ENV. Так как эти инструкции встречаются повсеместно, то введение *dynenv* скажется на производительности даже тех частей программы, где динамические переменные вовсе не используются. Очень важным является принцип, по которому за услуги платит только тот, кто их заказывает; следовательно, если какие-либо возможности языка не используются, то они не должны вызывать накладных расходов. С этой точки зрения введение дополнительного регистра является плохой идеей.

Поэтому вместо добавления общего для всех регистра мы реализуем иной подход, позволяющий сохранять необходимую информацию только тем, кто нуждается в ней и действительно будет пользоваться динамическими переменными. Конечно, расходы при этом будут немного выше, но они никогда не падут на непричастных. Динамические переменные будут храниться в особым образом сформированных фреймах стека, подобно переходам (см. рисунок 7.5). Функция search-dynenv-index ищет в стеке самый верхний фрейм,

содержащий динамическую переменную, — именно эту информацию должен был хранить регистр `*dynenv*`.

```
(define dynenv-tag (list '*dynenv*))

(define (search-dynenv-index)
  (let search ((i (- *stack-index* 1)))
    (if (< i 0) i
        (if (eq? (vector-ref *stack* i) dynenv-tag)
            (- i 1)
            (search (- i 1)) ) ) ) )

(define (pop-dynamic-binding)
  (stack-pop)
  (stack-pop)
  (stack-pop)
  (stack-pop) )

(define (push-dynamic-binding index value)
  (stack-push (search-dynenv-index))
  (stack-push value)
  (stack-push index)
  (stack-push dynenv-tag) )
```

Единственная не совсем очевидная деталь — это как именно компилировать обращения к динамическим переменным. Очевидно, что просто так у нас не получится скомпилировать выражение `(dynamic foo)` в байты, потому что символ `foo` — это не число. Та же проблема у нас была с глобальными переменными, тогда мы стали их нумеровать по мере появления. Почему бы не поступить аналогично и сейчас? Этим будет заниматься функция `get-dynamic-variable-index`:

```
(define *dynamic-variables* '())

(define (get-dynamic-variable-index n)
  (let ((where (memq n *dynamic-variables*)))
    (if where (length where)
        (begin
          (set! *dynamic-variables* (cons n *dynamic-variables*))
          (length *dynamic-variables*) ) ) ) )
```

Каждой динамической переменной ставится в соответствие индекс, по которому её можно отыскать в окружении динамических переменных, располагаемом в стеке. Переменная `*dynamic-variables*` в принципе нужна только для компиляции, но возможность вывести имя переменной в случае ошибки — тоже вещь хорошая. Определение функции `find-dynamic-value` оставляется вам в качестве упражнения.

В этом разделе приведена реализация динамических переменных с помощью пары специальных форм. Функциональная реализация лучше соотносится с духом Scheme, но она не полностью эквивалентна текущей. Аргументы функций могут быть как непосредственными значениями, так и выражениями, тогда как специальным формам позволяется вводить особые требования к синтаксису. Очевидно, что мы не сможем заранее пронумеровать динамические переменные, если их поддержка реализуется с помощью функций, так как это даёт возможность вычислять символы во время исполнения программы. [**см. стр. 72**] В этом случае уже придётся, по-видимому, как-то использовать сами символы напрямую, но поддержка модулей, пространств имён и различные оптимизации могут повлиять на семантику сравнения символов, что в свою очередь скажется на динамических переменных.

7.9. Исключения

В любом сколь-нибудь развитом языке программирования обязательно присутствует система обработки ошибок. Возможность обработки ошибок — это ключ к написанию устойчивых, надёжных приложений. К сожалению, в стандарте R^{5}RS подобная система не описана, поэтому давайте создадим её самостоятельно; просто чтобы показать, насколько она может быть полезной и удобной.

Понятие ошибки имеет несколько значений. Во-первых, это предотвратимые, но почему-то не предусмотренные ситуации. Например, ошибки типизации: `(car 33)`, недопустимые значения аргументов: `(quotient 111 0)`, или их неправильное количество: `(cons)`. Подобных логических ошибок можно избежать, вовремя выполняя соответствующие проверки. Другой класс ошибок составляют непредотвратимые, исключительные ситуации, например, ошибка открытия файла. Для их обработки необходимы следующие механизмы:

1) Способ создания объекта, описывающего ошибку и поддающегося программной обработке.

2) Возможность передать этот объект определённой функции, которую пользователь назначил обработчиком ошибок.

Если в языке присутствуют эти механизмы, то пользователи языка закономерно захотят их применять. Так ошибки становятся *исключениями*, и появляется возможность описывать логику программ с их помощью: реализовывать только корректное, нормальное поведение, а всевозможные аномальные прерывания вычислений оставлять на откуп обработчику исключительных ситуаций.

Существует несколько моделей обработки исключений. В основном они базируются на идее динамического времени жизни. Если мы хотим обрабатывать ошибки, возникающие во время определённых вычислений, то очевидно,

что на время проведения этих вычислений с ними связывается определённая функция-обработчик. Если вычисления выполнились без проблем, то обработчик становится ненужным и автоматически утилизируется. В противном случае создаётся объект, описывающий произошедшую ошибку, и к нему применяется обработчик. Если ошибку первым заметил пользовательский код, то он же её и описывает. Если же ошибка была обнаружена системой, то возникает системное исключение.

Некоторые модели обработки поддерживают возможность возобновления прерванных вычислений. В случае возникновения некоторых ошибок (в частности, предусмотренных заранее программистом, вроде вызываемых формой `cerror` в Common Lisp) можно попытаться повторить вычисления ещё раз, начиная с того места, где они были прерваны. Во многих других случаях, к сожалению, единственным возможным и наиболее адекватным поведением будет переход куда-нибудь в безопасное место, дабы не усугублять и так непростую ситуацию.

Модель исключений Common Lisp более чем полна, но её масштабность не особо сочетается с данной книгой, чьей задачей является рассмотрение сути вещей. Модель ML не поддерживает возобновление после ошибок; если где-то возникает исключение, то это вызывает необратимую раскрутку стека до ближайшего обработчика. Возможно только перебросить исключение дальше, раскручивая стек до следующего обработчика. В нашем случае подобная модель не подходит, так как при возникновении ошибки теряется динамическое окружение переменных, а ведь проблема могла быть именно в нём. Поэтому мы пойдём другим путём. (На его выбор сильное влияние оказал EuLisp.) Сначала мы неформально опишем идею, а затем плавно перейдём к её реализации.

Специальная форма `monitor` устанавливает функцию-обработчик исключений на время вычислений, представляемых её телом. Таким образом, она имеет следующий синтаксис:

`(monitor` *обработчик формы*`...)`

Соответственно, в синтаксическом анализаторе `meaning` появляется новое правило:

```
... ((monitor) (meaning-monitor (cadr e) (cddr e) r tail?)) ...
```

Обработчик вычисляется и становится текущим обработчиком исключений. *Формы*, составляющие тело `meaning`, последовательно вычисляются внутри неявной формы `begin`. Если во время вычислений ничего плохого не произошло, то `monitor` возвращает значение последней вычисленной формы и устанавливает обратно предыдущий обработчик.

Если же возникает исключительная ситуация, то сначала отыскивается текущий обработчик, то есть обработчик, ассоциированный с динамически ближайшей формой `monitor`. (На машинном языке это читается как «ближайший к верхушке стека».) Найденная функция вызывается с двумя аргументами: логическим значением, показывающим, может ли она возобновить

вычисления после завершения обработки, и объектом, описывающим проблему. Вызов осуществляется специальным образом, оставляя стек неизменным. Естественно, в самом обработчике также могут возникнуть ошибки — за них отвечает предыдущий обработчик. Вычисления, выполняемые при обработке, могут или нормально вернуть значение, или же прерваться переходом. Если исключение является возобновляемым, то возвращённое значение становится значением формы, вызвавшей исключение. Если возобновление запрещено, а значение всё равно возвращается, то возникает новое (невозобновляемое) исключение, сигнализирующее об этой проблеме. Если же обработка заканчивается переходом, то наблюдение за ошибками автоматически передаётся обработчику, ближайшему к новой текущей верхушке стека. В самом низу стека находится терминальный обработчик исключений. Если по каким-либо причинам исключение доходит до него, то он останавливает вирутальную машину и возвращает управление операционной системе.

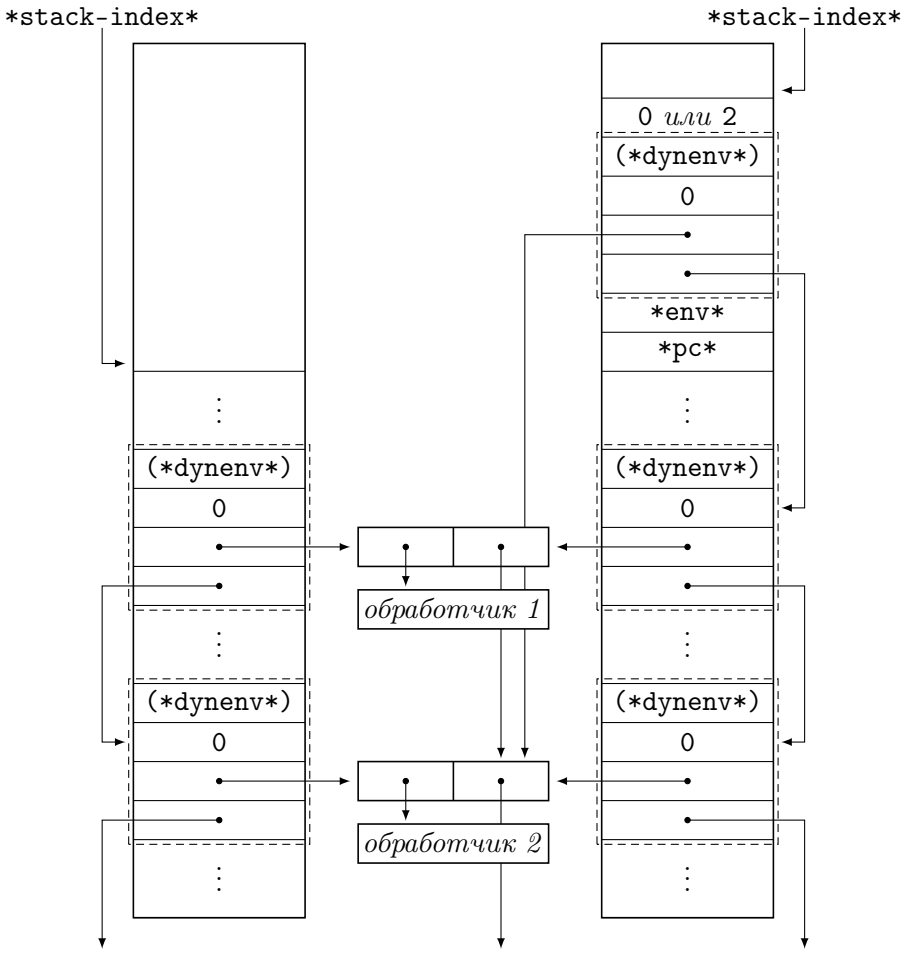


Рис. 7.6. Процесс обработки исключения.

Предобработка превращает форму `monitor` в комбинацию двух новых инструкций, отвечающих за работу со стеком обработчиков исключений:

```
(define (meaning-monitor e e+ r tail?)
  (let ((m (meaning e r #f))
        (m+ (meaning-sequence e+ r #f)) )
    (append m (PUSH-HANDLER) m+ (POP-HANDLER)) ) )

(define (PUSH-HANDLER) (list 246))
(define (POP-HANDLER)  (list 247))
```

Для обработки случаев возобновления невозобновляемого вводится дополнительная инструкция:

```
(define (NON-CONT-ERR) (list 245))
```

Инструкции `PUSH-HANDLER` и `POP-HANDLER` можно реализовать по-разному, так что сразу выделим их в отдельные функции:

```
(define-instruction (PUSH-HANDLER) 246
  (push-exception-handler) )

(define-instruction (POP-HANDLER) 247
  (pop-exception-handler) )
```

Вот теперь примемся за работу. Определить, какой из обработчиков ближе всех к верхушке стека, не составляет проблем: это типичная задача, решаемая с помощью динамических переменных. Заодно мы получаем приятный подарок: под стек обработчиков не потребуется специальный регистр. Вместо этого обработчики можно расположить в динамической переменной с индексом `0`, так как функция `get-dynamic-variable-index` не может выдать этот индекс обычным динамическим переменным. Хитрость здесь лишь в том, как сделать так, чтобы при выполнении текущего обработчика возникающие в нём ошибки перенаправлялись предыдущему. Эта проблема решается связыванием в список значений динамических переменных, хранимых по индексу `0`, что даёт возможность получить доступ к предыдущему обработчику, не снимая со стека текущий.

```
(define (search-exception-handlers)
  (find-dynamic-value 0) )

(define (push-exception-handler)
  (let ((handlers (search-exception-handlers)))
    (push-dynamic-binding 0 (cons *val* handlers)) ) )

(define (pop-exception-handler)
  (pop-dynamic-binding) )
```

Если вычисления проходят гладко, то находящиеся в стеке обработчики не вмешиваются в работу программы. Но при возникновении исключительной

ситуации вызывается функция `signal-exception`, которая является заменой старой доброй `wrong`.

```
(define (signal-exception continuable? exception)
  (let ((handlers (search-exception-handlers))
        (v* (allocate-activation-frame (+ 2 1))) )
    (set-activation-frame-argument! v* 0 continuable?)
    (set-activation-frame-argument! v* 1 exception)
    (set! *val* v*)
    (stack-push *pc*) (preserve-environment)
    (push-dynamic-binding 0 (if (null? (cdr handlers)) handlers
                                (cdr handlers) ))
    (if continuable?
      (stack-push 2)   ; выполнится (POP-HANDLER) (RESTORE-ENV) (RETURN)
      (stack-push 0) ) ; выполнится (NON-CONT-ERR)
    (invoke (car handlers) #t) ) )
```

Функция `signal-exception` принимает два аргумента: флаг, указывающий, допускается ли возобновление выполнения после успешного завершения обработчика, и значение, описывающее исключение. Это значение может быть абсолютно любым: например, предопределённые исключения представляются списками из человекочитаемой строки и дополнительных данных. Common Lisp и EuLisp допускают передачу только объектов специальных классов.

Обработка исключения происходит так. Сначала из динамического окружения достаётся список активных обработчиков, затем вручную формируется запись активации, которая будет передана первому из них. После этого специальным образом подготавливается стек, так как обработчику для правильной работы необходимо передать некоторую информацию, которой не место в записи активации. А именно, в стек заталкиваются текущее состояние счётчика команд и окружение (на случай, если придётся возвращаться). Далее, текущий обработчик исключений хитрым образом заменяется предыдущим: в динамической переменной сохраняется урезанный список активных обработчиков, что позволяет элегантно организовать передачу обязанностей предыдущего обработчика следующему. Так как в этом списке всегда должен оставаться как минимум один обработчик, то последний из них никогда не отбрасывается. Естественно, этот терминальный обработчик сам не должен вызывать исключений; и это действительно так, потому что он лишь сообщает об ошибке и завершает исполнение программы — тут нечему ломаться.

Итак, как же реализуется возможность возобновления управления после завершения обработки? Существует простое решение, но оно требует наличия в памяти определённых инструкций, расположенных по строго определённым абсолютным адресам. Мы уже расположили `FINISH` именно таким образом в прологе, почему бы не добавить туда ещё немного кода?

```
(define (code-prologue)
  (set! finish-pc 1)
  (append (NON-CONT-ERR) (FINISH) (POP-HANDLER) (RESTORE-ENV) (RETURN)) )
```

Теперь проблема возврата решается просто: перед вызовом обработчика достаточно положить в стек необходимый адрес возврата. Таким образом, `RETURN`, выполненная функцией-обработчиком, перейдёт в ту или иную ветку. Адрес `0` приводит к исполнению инструкции `NON-CONT-ERR`, вызывающей новое исключение; адрес `2` приводит к очистке стека от информации об исключении, которую `signal-exception` туда положила, восстановлению сохранённого ранее окружения и возврату управления назад в то место, где произошла ошибка.

```
(define-instruction (NON-CONT-ERR) 245
  (signal-exception #f (list "Non continuable exception continued")) )
```

Вот и всё, осталось только показать, как запускать виртуальную машину с учётом данных нововведений. Функция `run-machine` теперь должна корректным образом устанавливать терминальный обработчик исключений. Его можно реализовать, например, так, чтобы он выводил состояние машины и останавливал её. Без лишних слов:

```
(define (run-machine stack-size pc code constants global-names dynamics)
  (define base-error-handler-primitive
    (make-primitive base-error-handler) )
  (set! sg.current (make-vector (length global-names) undefined-value))
  (set! sg.current.names global-names)
  (set! *constants*         constants)
  (set! *dynamic-variables* dynamics)
  (set! *code*              code)
  (set! *env*               sr.init)
  (set! *stack*             (make-vector stack-size))
  (set! *stack-index*       0)
  (set! *val*               'anything)
  (set! *fun*               'anything)
  (set! *arg1*              'anything)
  (set! *arg2*              'anything)
  (push-dynamic-binding 0 (list base-error-handler-primitive))
  (stack-push finish-pc)
  (set! *pc* pc)
  (call/cc (lambda (exit)
             (set! *exit* exit)
             (run) )) )

(define (base-error-handler)
  (show-registers "Panic error! Contents of registers:")
  (wrong "Abort") )
```

Функция `signal-exception` легко превращается в доступный пользователям языка примитив, что позволяет использовать её (вместе с `monitor`) подобно `cerror`/`error` в Common Lisp и EuLisp для обработки пользовательских исключений. Однако, ввиду относительно высокой стоимости вызова, стоит

ограничить применения данного механизма действительно *исключительными* ситуациями.

В этом разделе была определена система обработки исключений. Рассмотренная модель позволяет вернуться из обработчика обратно и попробовать выполнить вычисления ещё раз. Также обработчик вызывается в том же динамическом окружении, где произошло исключение, что даёт больше возможностей для определения причин возникшей аномалии. Например, с помощью данной системы мы можем написать следующую весьма нестандартную версию факториала:

```
(monitor (lambda (c e) ((dynamic foo) 1))
  (let fact ((n 5))
    (if (= n 0) (/ 11 0)
        (* n (bind-exit (k)
               (dynamic-let (foo k)
                 (fact (- n 1)) ) )) ) ) )
```

Не следует вызывать переходы из ошибочного контекста в попытке быстро исправить ситуацию. Лучше сообщить об ошибке с помощью исключения и иметь в результате больше доступной информации о проблеме.

В силу того, что наша виртуальная машина — это лишь интерпретатор байт-кодов, система обработки исключений получилась относительно простой. Дело в том, что у нас ошибки могут возникнуть только в процессе исполнения какой-либо инструкции, а сами инструкции являются независимыми друг от друга. По крайней мере, состояние машины чётко определено в промежутках времени между исполнением инструкций. Для реальной машины это отнюдь не так и существует множество возможностей оставить её в неопределённом состоянии, например, какое-нибудь неожиданно полученное асинхронное прерывание.

У вас может возникнуть вопрос, почему некоторые предопределённые исключения являются возобновляемыми, а другие нет. Как уже было сказано, виртуальная машина работает с завидной аккуратностью, так что возобновить исполнение программы принципиально возможно после любого исключения, так как у нас всегда будет на руках точное значение счётчика команд. И снова, всё гораздо усложняется в реальности, где процессоры могут переупорядочивать исполняемые инструкции, а какое-нибудь деление на ноль, например, может быть обработано только постфактум. Безопаснее всего предполагать, что только для пользовательских исключений можно обеспечить переносимую и корректную логику работы после возобновления исполнения прерванного кода. Маловероятно, что программа сможет исправить ошибку, возникшую в самой системе.

Кстати, с этим связан ещё один интересный момент. Вспомните, что все ошибки предобработки теперь вызывают `signal-exception`, иногда даже допуская возобновление. И в этом действительно есть определённый смысл:

компилятор, используя специальный обработчик исключений, получает возможность самостоятельно исправлять некоторые недосмотры предобработки на лету.

7.10. Раздельная компиляция

Программы на языках вроде Си и Паскаля обычно компилируются и сохраняются в файлах. Лисп ничем не хуже, так что в этом разделе мы создадим подобный компилятор, а также компоновщик, собирающий отдельные кусочки кода воедино, и загрузчик, запускающий полученные программы.

7.10.1. Компилируем файлы

Откомпилировать программу, хранящуюся в файле, проще простого. Этим занимается функция `compile-file`, показанная ниже. Сначала она инициализирует списки глобальных переменных `g.current` и цитат `*quotations*`, а также список `*dynamic-variables*`. Затем выражения, считанные из файла, компилируются так, как будто бы все они находятся в огромной форме `begin`, а результат компиляции записывается в другой файл вместе со значениями трёх вышеупомянутых переменных.

```
(define (read-file filename)
  (call-with-input-file filename
    (lambda (in)
      (let gather ((e (read in)) (content '()))
        (if (eof-object? e)
            (reverse content)
            (gather (read in) (cons e content)) ) ) ) ) )

(define (compile-file filename)
  (set! g.current '())
  (set! *quotations* '())
  (set! *dynamic-variables* '())
  (let* ((complete-filename (string-append filename ".scm"))
         (e                 `(begin . ,(read-file complete-filename)))
         (code              (make-code-segment (meaning e r.init #t)))
         (global-names      (map car (reverse g.current)))
         (constants         (apply vector *quotations*))
         (dynamics          *dynamic-variables*)
         (ofilename         (string-append filename ".so")) )
    (write-result-file ofilename
                       (list ";;; Bytecode object file for "
                             complete-filename )
                       dynamics global-names constants code
                       (length (code-prologue)) ) ) )
```

```
(define (write-result-file ofilename comments dynamics
                           global-names constants code entry )
  (call-with-output-file ofilename
    (lambda (out)
      (for-each (lambda (comment) (display comment out))
                comments ) (newline out) (newline out)

      (display ";;; Dynamic variables" out) (newline out)
      (write dynamics out) (newline out) (newline out)

      (display ";;; Global modifiable variables" out) (newline out)
      (write global-names out) (newline out) (newline out)

      (display ";;; Quotations" out) (newline out)
      (write constants out) (newline out) (newline out)

      (display ";;; Bytecode" out) (newline out)
      (write code out) (newline out) (newline out)

      (display ";;; Entry point" out) (newline out)
      (write entry out) (newline out) ) ) )
```

Дабы не завязнуть в трясине различий файловых систем и способов записи путей, предполагается, что имена файлов записываются простыми строками. Компилятор понимает переданную ему строку как полное название файла. (Под названием подразумевается часть имени без обычного расширения `.scm`.) Программа читается из `*.scm`-файла, а результат компиляции сохраняется в `*.so`-файл. Этот результат состоит из байт-кода, списка глобальных изменяемых переменных, списка динамических переменных, вектора цитат и точки входа — начального значения счётчика команд. Всё это записывается в файл без особых изысков с помощью стандартной функции `write`. Также там оставляется немного комментариев, чтобы облегчить чтение результата человеком.

Итого, компиляция такого исходного файла:

si/example.scm

```
(set! fact
      ((lambda (fact) (lambda (n)
                        (if (< n 0)
                            "Toctoc la tete!"
                            (fact n fact (lambda (x) x)) ) ))
       (lambda (n f k)
         (if (= n 0)
             (k 1)
             (f (- n 1) f (lambda (r) (k (* n r)))) ) ) ) )
```

создаёт следующий объектный файл:

si/example.so

```
;;; Bytecode object file for si/example.scm

;;; Dynamic variables
()

;;; Global modifiable variables
(FACT)

;;; Quotations
#("Toctoc la tete!")

;;; Bytecode
#(245 20 247 38 43 40 30 59 74 32 1 34 81 35 106 31 10 3 34 82 34 51
  60 39 46 30 39 2 34 1 34 82 35 105 34 2 34 40 30 19 72 32 6 1 2 34
  6 1 0 34 1 35 109 34 51 60 39 46 43 34 53 62 61 60 39 46 43 34 51
  60 32 40 30 38 72 32 1 34 81 35 107 31 4 9 0 30 24 6 1 0 34 1 34 6
  1 0 34 40 30 4 72 32 1 43 34 53 62 61 60 39 46 43 33 27 0 43)

;;; Entry point
5
```

7.10.2. Собираем приложение

Возможность компилировать отдельные файлы — это, конечно, здорово, но неплохо было бы запускать всё приложение целиком! Вторая утилита, которую мы создадим, соответствует тому, что называется *компоновщиком* (линкером, `ld` в Un⋆x). Её задачей является связывание нескольких отдельных объектных файлов в единый исполнимый файл. Функция `build-application` принимает имя результирующего исполнимого файла, а затем имена компонуемых объектных файлов.

```
(define (build-application application-name ofilename . ofilenames)
  (set! sg.current.names    '())
  (set! *dynamic-variables* '())
  (set! sg.current          (vector))
  (set! *constants*         (vector))
  (set! *code*              (vector))
  (let install ((filenames (cons ofilename ofilenames))
                (entry-points '()) )
    (if (pair? filenames)
        (let ((ep (install-object-file! (car filenames))))
          (install (cdr filenames) (cons ep entry-points)) )
        (write-result-file application-name
                           (cons ";;; Bytecode application containing "
                                 (cons ofilename ofilenames) )
```

```
*dynamic-variables*
sg.current.names
*constants*
*code*
entry-points ) ) ) )
```

Большая часть работы поручается функции `install-object-file!`, которая формирует пять переменных, необходимых для обеспечения корректной работы машины:

- `sg.current.names` — имена изменяемых глобальных переменных;
- `*dynamic-variables*` — имена динамических переменных;
- `sg.current` — значения изменяемых глобальных переменных;
- `*constants*` — значения цитат;
- `*code*` — вектор байт-кодов.

После сборки всех объектных файлов остаётся только создать записать результат в исполнимый файл. Его формат совпадает с форматом объектных файлов во всём, кроме точки входа: теперь это будет список точек входа.

Функция `install-object-file!` добавляет один скомпилированный файл в приложение и возвращает адрес его первой инструкции в едином векторе байт-кодов. Дописать немного кода в вектор `*code*` просто. Гораздо сложнее защитить то, что является личной собственностью отдельных файлов, и разделить между всеми файлами то, что должно быть общим. В следующем коде эту нелёгкую работу выполняют функции, начинающиеся на `relocate`.

```
(define (install-object-file! filename)
  (let ((ofilename (string-append filename ".so")))
    (if (probe-file ofilename)
        (call-with-input-file ofilename
          (lambda (in)
            (let* ((dynamics     (read in))
                   (global-names (read in))
                   (constants    (read in))
                   (code         (read in))
                   (entry        (read in)) )
              (close-input-port in)
              (relocate-globals! code global-names)
              (relocate-constants! code constants)
              (relocate-dynamics! code dynamics)
              (+ entry (install-code! code)) ) ) )
        (signal-exception #f
          (list "No such file" ofilename) ) ) ) )
```

```
(define (install-code! code)
  (let ((start (vector-length *code*)))
    (set! *code* (vector-append *code* code))
    start ) )
```

Цитаты, например, у каждого файла свои собственные, они не должны совпадать даже физически. Но в каждом файле они заново нумеруются с нуля, что обязательно приведёт к коллизиям, если ничего не предпринять. Общий учёт используемых цитат ведётся в переменной `*constants*`, но новые цитаты добавляются туда не сразу, а после аккуратного обновления их номеров, являющихся адресами соответствующих значений в общем векторе. Номер цитаты, как вы помните, располагается сразу же за кодом инструкции `CONSTANT`. Следующая функция пробегает по всему вектору байт-кодов, отыскивает все нужные инструкции и заменяет старые адреса новыми:

```
(define CONSTANT-code 9)

(define (relocate-constants! code constants)
  (define n (vector-length *constants*))
  (let ((code-size (vector-length code)))
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (= instr CONSTANT-code)
            (let* ((i (vector-ref code (+ pc 1)))
                   (quotation (vector-ref constants i)) )
              (vector-set! code (+ pc 1) (+ n i)) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) )
  (set! *constants* (vector-append *constants* constants)) )
```

Глобальные переменные, напротив, должны разделяться. Два файла, использующих `foo`, должны обращаться к одной и той же переменной `foo`. Эти переменные нумеруются в каждом файле независимо, поэтому при компоновке необходимо согласовать их адреса. Эти адреса представляются числами, следующими за инструкциями `GLOBAL-REF`, `CHECKED-GLOBAL-REF` и `SET-GLOBAL!`. Каждое найденное число соответствует имени переменной; этому же имени соответствует другое число — адрес переменной в собираемой программе. Остаётся только заменить везде первое вторым.

```
(define GLOBAL-REF-code         7)
(define CHECKED-GLOBAL-REF-code 8)
(define SET-GLOBAL!-code       27)

(define (relocate-globals! code global-names)
  (define (get-index name)
    (let ((where (memq name sg.current.names)))
      (if where (- (length where) 1)
          (begin (set! sg.current.names (cons name sg.current.names))
                 (get-index name) ) ) ) )
```

```
  (let ((code-size (vector-length code)))
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (or (= instr GLOBAL-REF-code)
                    (= instr CHECKED-GLOBAL-REF-code)
                    (= instr SET-GLOBAL!-code) )
            (let* ((i (vector-ref code (+ pc 1)))
                   (name (list-ref global-names i)) )
              (vector-set! code (+ pc 1) (get-index name)) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) )
  (let ((v (make-vector (length sg.current.names) undefined-value)))
    (vector-copy! sg.current v 0 (vector-length sg.current))
    (set! sg.current v) ) )
```

Аналогично поступаем с динамическими переменными:

```
(define DYNAMIC-REF-code 240)
(define DYNAMIC-PUSH-code 242)

(define (relocate-dynamics! code dynamics)
  (for-each get-dynamic-variable-index dynamics)
  (let ((dynamics (reverse dynamics))
        (code size (vector-length code)) )
    (let scan ((pc 0))
      (when (< pc code-size)
        (let ((instr (vector-ref code pc)))
          (when (or (= instr DYNAMIC-REF-code)
                    (= instr DYNAMIC-PUSH-code) )
            (let* ((i (vector-ref code (+ pc 1)))
                   (name (list-ref dynamics (- i 1))) )
              (vector-set! code (+ pc 1)
                           (get-dynamic-variable-index name) ) ) )
          (scan (+ pc (instruction-size code pc))) ) ) ) ) )
```

Обратите внимание на то, как и почему здесь используется `instruction-size`. Также стоит отметить, что глупо просматривать вектор кода три раза подряд, вполне можно выполнить всю работу и за один проход.

Выбранный формат представления скомпилированных программ позволяет множество вариаций. Например, можно понимать список глобальных изменяемых переменных как интерфейс к модулю, которым является файл. Такой модуль экспортирует все свои глобальные переменные для внешнего использования под теми именами, которые они имеют внутри данного модуля. Однако, нехорошо выставлять напоказ всё подряд, так что вдобавок к этому можно, например, разрешить экспортировать не все переменные или же дать возможность переименовывать их при экспорте. Можно даже представить себе специализированный язык для описания способов сборки модулей, управления именованием переменных и т. д. Поясним эту идею на примере языка, пред-

лагаемого в [QP91a] для EuLisp. Следующая директива описывает экспортируемые переменные модуля `mod`:

```
(ordered-union
  (only (fact) (expose "fact"))
  (union (except-pattern ("fib*") (expose "fib"))
         (rename ((call/cc call-with-current-continuation))
           (expose "scheme") )
         (expose "numeric") ) )
```

Предположим, запись `foo@mod` означает переменную с именем `foo` в модуле `mod`. Также допустим, что модуль `fact` определяет переменные `fact` и `fact100` (последняя содержит значение `(fact 100)`, которое очень часто бывает необходимым); модуль `fib` определяет переменные `fib`, `fib20` и `Fibonacci`. Модуль `numeric` среди множества определяемых им функций содержит также `fact` и `fib`, а модуль `scheme` определяет все стандартные функции R^5RS. Тогда создаваемый данной директивой модуль состоит из следующих частей: переменной `fact@fact` (а переменная `fact100@fact`, экспортируемая директивой `(expose "fact")`, отбрасывается директивой `only`); переменной `Fibonacci@fib` (все остальные фильтруются по шаблону с помощью `except-pattern`). Экспортируемая переменная `call-with-current-continuation@scheme` становится просто `call/cc@scheme`. Так как объединение, определяющее `mod`, объявлено как упорядоченное (ordered union), то модуль `fact` загружается перед остальными (`fib`, `numeric` и `scheme`), порядок обработки которых не указывается, но должен соответствовать их определениям. Например, модуль `numeric` наверняка использует возможности `scheme`, поэтому его следует обрабатывать одним из последних.

7.10.3. Запускаем приложение

Исполнимый файл создаётся, конечно же, для того, чтобы его исполняли. Благо, все сложности уже позади и написать загрузчик не составляет труда. Нужна только внимательность, чтобы не забыть проинициализировать какой-нибудь регистр.

```
(define (run-application stack-size filename)
  (if (probe-file filename)
      (call-with-input-file filename
        (lambda (in)
          (let* ((dynamics     (read in))
                 (global-names (read in))
                 (constants    (read in))
                 (code         (read in))
                 (entry-points (read in)) )
            (close-input-port in)
            (set! sg.current.names    global-names)
            (set! *dynamic-variables* dynamics)
```

```
        (set! sg.current (make-vector (length sg.current.names)
                                      undefined-value ))
        (set! *constants*         constants)
        (set! *code*              (vector))
        (install-code! code)
        (set! *env*               sr.init)
        (set! *stack*             (make-vector stack-size))
        (set! *stack-index*       0)
        (set! *val*               'anything)
        (set! *fun*               'anything)
        (set! *arg1*              'anything)
        (set! *arg2*              'anything)
        (push-dynamic-binding
         0 (list (make-primitive (lambda ()
                                   (show-exception)
                                   (*exit* 'aborted) ))) )
        (stack-push 1)
        (if (pair? entry-points)
            (for-each stack-push entry-points)
            (stack-push entry-points) ) )
      (set! *pc* (stack-pop))
      (call/cc (lambda (exit)
                 (set! *exit* exit)
                 (run) )) ) )
  (static-wrong "No such file" filename) ) )
```

Прежде всего, конечно же, необходимо прочитать файл с загружаемой программой. Входной порт закрывается сразу же, как становится ненужным. Далее, подобно `run-machine`, инициализируются переменные состояния машины. После установки терминального обработчика исключений в стек заталкиваются точки входа всех объектных файлов, составляющих приложение. Такой подход позволяет запускать единичный объектный файл аналогично цельному приложению. Стандартный обработчик исключений останавливает исполнение программы при возникновении исключительной ситуации; для этого захватывается продолжение функции `run-application` и сохраняется в переменной `*exit*`. Функция `show-exception` отвечает за вывод понятных и выразительных[11] сообщений об ошибках на основе дескриптора исключения, находящегося в тот момент в регистре `*val*`. Остаётся только не забыть об инструкции `RETURN` в конце каждого объектного файла, которая обеспечивает передачу управления следующему файлу в цепочке.

Функция `run-application` принимает максимальный размер стека как аргумент. Это отдельная и весьма важная проблема реализации: не допустить переполнения стека. Было бы слишком дорого выполнять проверку границ при каждом вызове `stack-push`. Иногда возможно использовать механизм

[11] Вроде `"Bus error: core not dumped"`.

страничной виртуальной памяти, предоставляемый операционной системой, чтобы избавиться от проверок и увеличивать размер стека при необходимости на лету. Наша эмуляция памяти вряд ли является эффективной, так как о стоимости выражения `(vector-ref` v i`)` нельзя сказать ничего определённого; здесь наверняка каждый раз будет выполняться проверка, что v это действительно вектор, а i является неотрицательным целым числом, меньшим, чем длина вектора v.

Функция `run-application` требует весьма немного приспособлений для исполнения скомпилированных приложений. Ей нужен всего лишь интерпретатор байт-кодов (`run`), а также функции для работы с векторами, списками и другими классами вроде `primitive` и `continuation`. Ещё потребуется функция `read` для чтения файлов. Всё это слабо зависит от конкретного компилятора, создающего объектные файлы (что, правда, не особо помогает при отладке), а ведь именно подобное отделение языка от исполнимых программ и есть целью компиляции!

7.11. Заключение

До полной реализации Scheme нашему компилятору не хватает множества функций стандартной библиотеки, а также поддержки оставшихся типов данных. С этим не должно возникнуть каких-либо принципиальных проблем, кроме разве что сложностей с реализацией проверки типов и допустимости значений.

Описанная реализация является довольно простой, но достигается эта простота игнорированием огромного числа всевозможных оптимизаций. Однако, основной упор в ней делается на том, чтобы использовать предыдущие наработки — оптимизированный интерпретатор, уже частично превращённый в компилятор. Действительно, существует глубокая связь между интерпретацией и компиляцией (рассмотренная подробнее в [Nei84]). Интерпретатор непосредственно исполняет программу, тогда как компилятор превращает программу в нечто, что будет исполнено потом. Следовательно, они отличаются лишь способом трактовки инструкций языка: исполнять ли соответствующие действия или генерировать соответствующий код. Мы воспользовались данной связью, чтобы легко превратить имеющийся интерпретатор в компилятор.

Основной недостаток такого подхода в том, что здесь используется лишь информация, сохранённая в промежуточном языке, а её явно недостаточно для качественной компиляции. Мы ничего не знаем, к примеру, об используемых локальных переменных; несомненно, это один из главных источников информации, необходимой для проведения различных оптимизаций. Мы не знаем, изменяются ли значения переменных, захватываются ли они замыканиями, захватываются ли несколькими или только одним, изменяются ли захваченные копии и т. д. Во многих случаях подобные статические сведения о переменных позволяют избавиться от записей активаций и пользоваться

одним только стеком — а это, очевидно, положительно сказывается на скорости работы. Записи активаций действительно полезны лишь для сохранения значений переменных, захваченных замыканиями; комбинаторы вполне могут обойтись и без них. Так как значения передаваемых аргументов уже находятся в стеке, то их можно там и оставить, не создавая при этом объектов-посредников и экономя таким образом значительное количество времени.

7.12. Упражнения

Упражнение 7.1 Реализуйте поддержку динамических переменных с помощью регистра `*dynenv*` для компилятора и виртуальной машины из этой главы. [**см. стр. 304**]

Упражнение 7.2 Определите функцию `load`, которая загружает и исполняет скомпилированные программы. Например, если у нас есть файл `fact.so`, полученный из файла `fact.scm`, то следующая программа должна успешно компилироваться и выдавать ожидаемое число 120:

```
(begin (load "fact")
       (fact 5) )
```

Упражнение 7.3 Реализуйте функцию `global-value`, принимающую имя глобальной переменной и возвращающую её текущее значение.

Упражнение 7.4 Переопределите инструкции, отвечающие за работу с динамическими переменными, реализовав их с помощью ближнего связывания. [**см. стр. 43**]

Упражнение 7.5 [**см. стр. 319**] Напишите функцию, переименовывающую экспортируемые глобальные переменные. Если `fact.so` это объектный файл, то следующая программа должна создать новый модуль `nfact.so`, где переменная `fact` имеет имя `factorial`:

```
(build-application-with-renaming-variables
 "fact.so" "nfact.so" '((fact factorial)) )
```

Упражнение 7.6 [**см. стр. 235**] Измените инструкцию `CHECKED-GLOBAL-REF` так, чтобы после определения соответствующей переменной она автоматически изменялась на `GLOBAL-REF`.

Проект 7.7 Многие реализации Лиспа и Scheme имеют компиляторы в байт-код. Возьмём, например, GNU Emacs Lisp [LLSt93] и xscheme [Bet91]. Реализуйте соответствующие виртуальные машины для них.

Рекомендуемая литература

Литературы на тему основ компиляции весьма немного. Собственно, это одна из причин появления данной книги. Как бы то ни было, вам стоит взглянуть на [All78] и [Hen80]. Об общих принципах компиляции функциональных языков неплохо написано в [Dil88]. В следующих книгах тоже есть несколько интересных компиляторов с комментариями: [Ste78, AS85, FWH92].

Глава 8

Вычисления и рефлексия

Уникальной чертой Лиспа является его вычислитель: `eval`. Хотя в данной книге вычисления упоминаются чуть ли не на каждой странице, мы никогда не касались возможности явного использования вычислителя. И на то есть веские причины: явные вычисления создают определённые затруднения с их формализацией, они влияют на целостность окружения вычислений, поднимают неудобные вопросы лингвистического плана. Всё это дало жизнь расхожему выражению: «`eval` is *evil*». Нахождение возможностей плодотворного применения `eval` — это первый шаг к рефлексивному программированию — теме, которая также будет затронута в этой главе.

Каждый рассмотренный нами интерпретатор показывал ту или иную сторону процесса вычислений. Для большинства из них явное предоставление вычислителя пользователям не требует значительных усилий и выполняется всего парой строк кода. Это тривиальная задача по сравнению с реализацией самого интерпретатора. Однозначно можно сказать [**см. стр. 20**], что облегчение её решения намеренно заложено в дизайн Лиспа. Явная функция `eval` присутствовала ещё в первых работах отцов-основателей: [McC60, MAE+62].

Наличие в языке явного вычислителя является его фундаментальным качеством, позволяющим реализовать мощную макросистему, объединить язык и среду разработки, а также наделить программы уже упомянутыми возможностями интроспекции. Конечно, при этом имеются и недостатки, такие как макросы, неотрывность среды разработки от языка и сующая свой нос куда попало интроспекция. Подобно волшебному джинну, `eval` может быть как полезной, так и принести погибель.

Итак, какой контракт нам стоит заключить с `eval`? Безусловно, одним из его пунктов должно быть утверждение

$$\texttt{(eval '}\pi\texttt{)} \equiv \pi \tag{1}$$

Вскоре мы покажем, что это уравнение далеко от однозначности.

8.1. Программы и значения

Давайте попробуем добавить `eval` в самый первый интерпретатор, рассмотренный в этой книге. [**см. стр. 22**] Он написан на чистом Scheme без использования каких-либо библиотек. Для начала сделаем `eval` специальной формой. Следовательно, в `evaluate` появляется соответствующая ветка:

```
(define (evaluate e env)
  (if (atom? e)
      (cond ((symbol? e) (lookup e env))
            ((or (number? e)(string? e)(char? e)
                 (boolean? e)(vector? e) )
              e)
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
        ((quote) (cadr e))
        ((if)     (if (evaluate (cadr e) env)
                      (evaluate (caddr e) env)
                      (evaluate (cadddr e) env) ))
        ((begin)  (eprogn (cdr e) env))
        ((set!)   (update! (cadr e) env (evaluate (caddr e) env)))
        ((lambda) (make-function (cadr e) (cddr e) env))
        ((eval)   (evaluate (evaluate (cadr e) env) env))
        (else     (invoke (evaluate (cadr e) env)
                          (evlis (cdr e) env) )) ) ) )
```

Специальная форма `eval` вычисляет свой единственный аргумент (прямо как функция), а затем ещё раз вычисляет полученное в результате значение в текущем окружении. К сожалению, в этом вроде как самоочевидном описании на самом деле полно неясностей.

Как уже не раз было сказано, `evaluate` разделяется на этап анализа и этап исполнения. В последних интерпретаторах мы даже явно ввели две отдельные функции: `meaning` и `run`. Функция `evaluate` принимает программы, а возвращает значения — результаты вычислений. Вот здесь и возникает первая проблема (назовём её проблемой типизации): в форме `(evaluate (evaluate ...) ...)` внешняя `evaluate` применяется к значению, а не к программе. Являются ли программы значениями? А значения программами?

Определения языков программирования обычно включают в себя грамматику — описание допустимых синтаксических конструкций этого языка. Грамматика Scheme приведена в [KCR98]. Там написано, что определённые наборы букв и скобочек являются синтаксически допустимыми программами. Кроме того, грамматика задаёт синтаксис значений (данных), поэтому можно легко проверить, является ли программа значением с синтаксической точки зрения. Благо, в Scheme есть универсальная функция `read`, умеющая читать как программы, так и данные. Ведь ничто не обязывает нас сразу же исполнять считанные `read` программы. Многие интерпретаторы Smalltalk, вроде [GR83],

сперва проверяют синтаксис программы и отображают её в специальном окне, подсвечивая все найденные синтаксические ошибки. Это достаточно распространённая практика, так что при необходимости вполне можно трактовать программы как значения, если с точки зрения синтаксиса они являются нормальными значениями.

Но вот обратное не всегда верно: существует множество значений, не являющихся синтаксически корректными программами, например, `(quote . 1)`. А есть и значения, которые нельзя однозначно отнести к той или иной группе.

- Возьмём значение, являющееся корректной программой во всём, кроме пары-тройки деталей. Например, `(if #t 1 (quote . 2))`. Это программа? Если передать данное выражение определённой ранее `eval`, то она вычислит его без особых проблем, так как не будет даже смотреть на `(quote . 2)`. Но данное выражение нарушает правила записи программ на Scheme.

- Или, например, значение, возвращаемое формой вроде `‘(’,car ’(a b))`, где первым термом является процитированное значение функции `car`. Это программа? В соответствии с синтаксисом R^{5}RS, это значение не является программой, так как у него нет *внешнего представления*: его нельзя непосредственно набрать на клавиатуре. Тем не менее, `eval` опять выполнит такую программу без запинки.

- Теперь воспользуемся нотацией Common Lisp: пусть `#1=` даёт имя следующему за ним выражению, а `#1#` позволяет сослаться на выражение с именем `1`. Держа это в голове, рассмотрим следующую программу, чьё графическое представление показано на рисунке 8.1.

  ```
  (let ((n 4))
    #1=(if (= n 1) 1
           (* n ((lambda (n) #1#) (- n 1))) ) )
  ```

  Это значение содержит цикл. Можно сказать, это синтаксически рекурсивная программа. К сожалению, синтаксис Scheme запрещает подобные трюки. Осталось только объяснить это `eval`, которую данный цикл ничуть не смущает.

Конечно, вам может показаться, что все эти примеры взяты из специальной книги «Тысяча и одно извращение на Лиспе», но зато они вполне доходчиво демонстрируют, насколько расплывчатой становится идея программы с появлением `eval`. Похожие проблемы возникают и у макросов, проводящих разнообразные вычисления со значениями-программами.

Кстати, вспомните наш разговор о статических и динамических ошибках. [**см. стр. 234**] Где именно проводить границу между ними? Наиболее либеральная позиция: допускать любые синтаксические аномалии вроде `(quote . 3)`, пока они не мешают проводить вычисления. Наиболее строгая: допускать

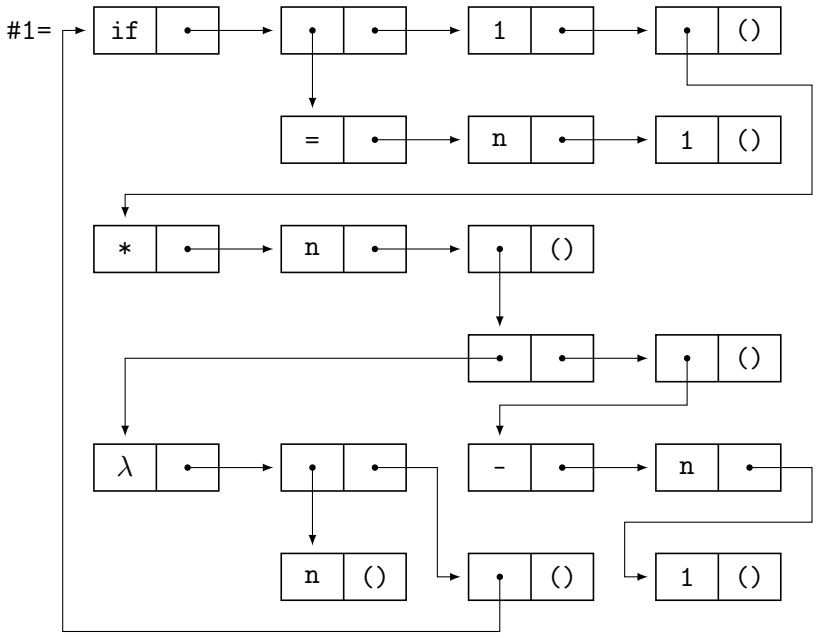


Рис. 8.1. Синтаксически рекурсивный факториал.

только такие программы, которые представимы в виде конечного ациклического графа синтаксически корректных выражений. В этом случае упомянутый ранее синтаксически рекурсивный факториал будет признан недопустимым, несмотря на то, что отдельные выражения, из которых он состоит, являются вполне корректными. Пожалуй, мы примем именно эту позицию (её же занимает и Scheme), потому как существует алгоритм, рассмотренный в [Que92a], позволяющий переписать произвольную синтаксически рекурсивную программу в виде дерева инструкций без циклов. Что касается цитат, то мы разрешим только цитаты, состоящие из конечного числа атомов, имеющих внешнее представление. Таким образом, под запретом цитирования оказываются замыкания, продолжения, порты, а также циклические[1] конструкции.

Правила, формализующие понятие синтаксически корректной программы, можно записать в виде следующих предикатов. Они проверяют синтаксис выражений и обнаруживают[2] циклы.

```
(define (program? e)
  (define (program? e m)
    (if (atom? e)
        (or (symbol? e) (number? e) (string? e) (char? e) (boolean? e))
        (if (memq e m) #f
```

[1] Вот именно их можно и разрешить: это не так сложно сделать, а они могли бы быть полезными, например, при реализации Meroonet.

[2] Как думаете, значение `(let ((e '(x))) '(lambda ,e ,e))` считается допустимой программой?

```
            (let ((m (cons e m)))
              (define (all-programs? e+ m+)
                (if (memq e+ m+) #f
                    (let ((m+ (cons e+ m+)))
                      (and (pair? e+)
                           (program? (car e+) m)
                           (or (null? (cdr e+))
                               (all-programs? (cdr e+) m+) ) ) ) ) )
              (case (car e)
                ((quote) (and (pair? (cdr e))
                              (quotation? (cadr e))
                              (null? (cddr e)) ))
                ((if) (and (pair? (cdr e))
                           (program? (cadr e) m)
                           (pair? (cddr e))
                           (program? (caddr e) m)
                           (pair? (cdddr e))
                           (program? (cadddr e) m)
                           (null? (cddddr e)) ))
                ((begin) (all-programs? (cdr e) '()))
                ((set!) (and (pair? (cdr e))
                             (symbol? (cadr e))
                             (pair? (cddr e))
                             (program? (caddr e) m)
                             (null? (cdddr e)) ))
                ((lambda) (and (pair? (cdr e))
                               (variables-list? (cadr e))
                               (all-programs? (cddr e) '()) ))
                (else (all-programs? e '())) ) ) ) ) )
  (program? e '()) )

(define (variables-list? v*)
  (define (variables-list? v* already-seen)
    (or (null? v*)
        (and (symbol? v*)
             (not (memq v* already-seen)) )
        (and (pair? v*)
             (symbol? (car v*))
             (not (memq (car v*) already-seen))
             (variables-list? (cdr v*)
                              (cons (car v*) already-seen) ) ) ) )
  (variables-list? v* '()) )

(define (quotation? e)
  (define (quotation? e m)
    (if (memq e m) #f
```

```
        (let ((m (cons e m)))
          (or (null? e) (symbol? e) (number? e)
              (string? e) (char? e) (boolean? e)
              (and (vector? e)
                   (let loop ((i 0))
                     (or (>= i (vector-length e))
                         (and (quotation? (vector-ref e i) m)
                              (loop (+ i 1)) ) ) ) )
              (and (pair? e)
                   (quotation? (car e) m)
                   (quotation? (cdr e) m) ) ) ) ) )
  (quotation? e '()) )
```

Вооружившись предикатом `program?`, можно уточнить определение специальной формы `eval` следующим образом:

```
... ((eval) (let ((v (evaluate (cadr e) env)))
              (if (program? v)
                  (evaluate v env)
                  (wrong "Illegal program" v) ) )) ...
```

Увы, это определение всё ещё весьма топорно, потому как предыдущие предикаты не проверяют, *является ли* значение программой или корректной цитатой; они лишь говорят, правильно ли *записана* программа или цитата. Это предел информативности, доступной значениям. Значения сами по себе — это ни программы, ни цитаты. Действительно, программа — это то, что интерпретатор считает программой, то же самое касается и цитат. Именно интерпретатор придаёт смысл значениям. Для уточнения этого момента перейдём к интерпретатору из четвёртой главы. [**см. стр. 142**] В нём память и продолжения используются явно, а значения интерпретируемого Scheme представляются объектами-замыканиями. Вот его вычислитель с добавленной формой `eval`:

```
(define (evaluate e r s k)
  (if (atom? e)
      (if (symbol? e) (evaluate-variable e r s k)
          (evaluate-quote e r s k) )
      (case (car e)
        ((quote)  (evaluate-quote (cadr e) r s k))
        ((if)     (evaluate-if (cadr e) (caddr e) (cadddr e) r s k))
        ((begin)  (evaluate-begin (cdr e) r s k))
        ((set!)   (evaluate-set! (cadr e) (caddr e) r s k))
        ((lambda) (evaluate-lambda (cadr e) (cddr e) r s k))
        ((eval)   (evaluate-eval (cadr e) r s k))
        (else     (evaluate-application (car e) (cdr e) r s k)) ) ) )

(define (evaluate-eval e r s k)
  (evaluate e r s
```

```
    (lambda (v ss)
      (let ((ee (transcode-back v s)))
        (if (program? ee)
            (evaluate ee r ss k)
            (wrong "Illegal program" ee) ) ) ) ) )
```

В этом интерпретаторе вычисленное значение первого аргумента формы `eval` дополнительно преобразуется (с помощью `transcode-back`) из внутреннего представления во внешнее. Затем проверяется правильность записи программы, после чего она исполняется. Переменные `e` и `ee` хранят описания программ, а переменная `v` — значения. Задача `transcode-back`: взять значение и что-то с ним сделать, чтобы его можно было считать корректным описанием программы. Примерно таким же путём можно получить `eval` в интерпретаторе, где нет `eval`, но есть функция `load`: записываем программу в файл, задачу `transcode-back` в этом случае выполняет `display`; читаем и исполняем программу из файла с помощью `load`. Таким образом, программа «компилируется»[3] в имя этого файла.

Наконец, давайте рассмотрим также быстрый интерпретатор из шестой главы. [**см. стр. 222**] Там обнаружится ещё несколько интересных моментов. В нём значения интерпретируемого языка представляются родными значениями Scheme, так что нужда в преобразованиях отпадает. Главной задачей этого интерпретатора было предварительное вычисление всех выражений, которые можно вычислить статически.

```
(define (meaning e r tail?)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e r tail?)
                      (meaning-quotation e r tail?) )
      (case (car e)
        ((quote)  (meaning-quotation (cadr e) r tail?))
        ((lambda) (meaning-abstraction (cadr e) (cddr e) r tail?))
        ((if)     (meaning-alternative (cadr e) (caddr e) (cadddr e)
                                       r tail?))
        ((begin)  (meaning-sequence (cdr e) r tail?))
        ((set!)   (meaning-assignment (cadr e) (caddr e) r tail?))
        ((eval)   (meaning-eval (cadr e) r tail?))
        (else     (meaning-application (car e) (cdr e) r tail?)) ) ) )

(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (lambda ()
      (let ((v (m)))
```

[3] Правда, такой способ определения `eval` позволяет её использовать только на глобальном уровне, воспользоваться локальным окружением функций загружаемые программы не смогут. К вопросам управления окружениями при явных вычислениях мы вернёмся чуть позже.

```
(if (program? v)
    (let ((mm (meaning v r tail?)))
      (mm) )
    (wrong "Illegal program" v) ) ) ) ) )
```

Вот теперь, на примере компилирующего интерпретатора, преобразующего программы в деревья из инструкций-замыканий, становится окончательно понятным, что компиляция `v` не может быть выполнена статически. Впервые узел дерева, создаваемый компилятором, не является простой операцией вроде выделения памяти, проверки условия или обращения к структурам данных. Вместо этого (о ужас!) внутри замыкания находится вызов `meaning`. Это значит, что во время исполнения программы требуется держать в памяти функцию `meaning`, а также всех её друзей, чтобы иметь возможность выполнять компиляцию программ на лету, — недешёвое удовольствие!

В этом разделе мы попытались уменьшить путаницу между понятиями значения и программы. Похожие отношения существуют также между цитатами и значениями. Поэтому иногда говорят, что `eval` и `quote` дополняют друг друга: `quote` получает описание некоторого значения, которое она должна создать, а `eval` получает значение, являющееся описанием некоторых действий, которые она должна выполнить. Важно понимать, что они обе преобразуют обозначения в обозначаемые объекты.

8.2. `eval` как специальная форма

`eval` в виде специальной формы приближает нас к идеалу — уравнению (1); то есть, `(eval (quote` π`))` становится эквивалентом π. Но это верно лишь при чисто алгебраическом понимании этого уравнения, как, например, в [FH89, Mul92]. Любая надёжная теория допускает взаимозаменимость двух объектов только в случае, когда они эквивалентны *в любом контексте*. Точнее говоря, получается, что (1) — это менее строгая форма уравнения (2), где C означает контекст вычислений:

$$\forall C\colon\ C[\texttt{(eval '}\pi\texttt{)}] \equiv C[\pi] \tag{2}$$

Следовательно, значением замкнутой формы (не имеющей свободных переменных) вроде `(eval '((lambda (x y) x) 1 2))` всегда будет `1`. Однако, вычисляемая форма может иметь контекст в виде лексического окружения:

```
((lambda (x) (eval 'x))
 3 )                          → 3
((lambda (x y) (eval y))
 4 'x )                       → 4
((lambda (x y z) (eval y))
 5 (list 'eval 'z) 'x )       → 5
```

Подобное поведение возможно только при условии, что `eval` доступно лексическое окружение. В быстром интерпретаторе замыканию, создаваемому `meaning` для `eval`, необходимо не только получить доступ к функции `meaning`, но и захватить текущее лексическое окружение `r` (а также `tail?`), чтобы проводить внутренние вычисления в правильном локальном лексическом окружении. Следовательно, во время исполнения программ потребуется хранить не только `meaning`, а ещё и всевозможные окружения.

Это видно ещё лучше на примере компилятора в байт-код из седьмой главы. [**см. стр. 269**] Функция-анализатор `meaning-eval` вызывает функцию-генератор `EVAL/CE` (от *eval in current environment*). Она принимает выражение, которое необходимо вычислить, и окружение для компиляции.

```
(define (meaning-eval e r tail?)
  (let ((m (meaning e r #f)))
    (EVAL/CE m r) ) )

(define (EVAL/CE m r)
  (append (PRESERVE-ENV) (CONSTANT r) (PUSH-VALUE)
          m (COMPILE-RUN) (RESTORE-ENV) ) )
```

Функция `EVAL/CE` игнорирует `tail?` и всегда сохраняет локальное окружение. Новая «простейшая» инструкция `COMPILE-RUN` содержит в себе весь компилятор-интерпретатор. Она забирает выражение из регистра `*val*` и окружение с верхушки стека, а затем применяет к ним функцию `compile-on-the-fly`, которая выполняет компиляцию, загружает в память полученный байт-код и передаёт ему управление так, как если бы там была обычная функция.

```
(define (COMPILE-RUN) (list 255))

(define-instruction (COMPILE-RUN) 255
  (let ((v *val*)
        (r (stack-pop)) )
    (if (program? v)
        (compile-and-run v r #f)
        (signal-exception #t
          (list "Illegal program" v) ) ) ) )

(define (compile-and-run v r tail?)
  (unless tail? (stack-push *pc*))
  (set! *pc* (compile-on-the-fly v r)) )

(define (compile-on-the-fly v r)
  (set! g.current '())
  (for-each g.current-extend! sg.current.names)
  (set! *quotations* (vector->list *constants*))
  (set! *dynamic-variables* *dynamic-variables*)
```

```
(let ((code (apply vector (append (meaning v r #f) (RETURN)))))
  (set! sg.current.names (map car (reverse g.current)))
  (let ((v (make-vector (length sg.current.names) undefined-value)))
    (vector-copy! sg.current v 0 (vector-length sg.current))
    (set! sg.current v) )
  (set! *constants* (apply vector *quotations*))
  (set! *dynamic-variables* *dynamic-variables*)
  (install-code! code) ) )
```

При компиляции на лету, естественно, потребуются все глобальные переменные, используемые при обычной компиляции: `g.current` (окружение изменяемых глобальных переменных), `*quotations*` и `*dynamic-variables*`.[4] Откомпилированный код заканчивается инструкцией `RETURN`, возвращающей управление обратно вызывающей программе. После завершения компиляции окружение исполнения расширяется новыми цитатами, а также глобальными и динамическими переменными, созданными в процессе. В конце концов полученный фрагмент кода загружается в память и начинается его исполнение.

Подобная загрузка принципиально аналогична механизму динамически подключаемых библиотек. Для языков вроде Си `eval` можно свести к записи строки с программой в файл, компиляции этого файла с помощью `cc`, загрузке средствами операционной системы полученной библиотеки и вызову содержащихся внутри неё функций. К сожалению, на данном этапе наша машина не поддерживает удаление из памяти загружаемого подобным образом кода, так что он остаётся висеть мёртвым грузом после завершения исполнения. Мусорить всегда легче, чем убирать.

Таким образом, стоимость новой специальной формы составляют:

- новая инструкция `COMPILE-RUN`, требующая наличия функции `meaning` и некоторых другими, что приводит к значительному увеличению размера кода для небольших приложений;
- расходы памяти на сохранение текущего окружения при каждом вызове `eval`.

К счастью, дополнительные окружения необходимо сохранять только при этих вызовах, так что суммарная стоимость использования `eval` линейно зависит от количества данных форм в программе.

Кроме того, `eval` иногда используется для отладки: ведь она позволяет внедрить REPL в саму программу и просматривать или изменять переменные изнутри, обращаясь к ним по именам. А если туда встроен ещё и отладчик, позволяющий прервать исполнение программы в произвольный момент времени (например, по команде с клавиатуры), то кроме окружений необходимо вдобавок где-то хранить и весь исходный код вместе с таблицей символов.

[4] В отличие от остальных, переменная `*dynamic-variables*` общая для компилятора и машины-исполнителя. Сейчас подобные присваивания ей бессмысленны, но они напоминают, что их необходимо проводить, если данная переменная не будет разделяемой.

8.2.1. Глобальные переменные

Согласно контракту (2), выражение `(eval '(set! foo 33))` должно быть эквивалентным `(set! foo 33)`. Впервые мы столкнулись с неоднозначностью семантики присваивания в четвёртой главе. [**см. стр. 142**] Затем в разделе 6.1.9 были рассмотрены различные варианты реализации окружений и присваивания. [**см. стр. 243**] Следовательно, если для создания переменной достаточно её упоминания, то и `eval` это позволено. И наоборот, если переменные необходимо определять до использования, то данное ограничение касается в том числе и `eval`. Функция `compile-on-the-fly` расширяет в конце своей работы глобальное окружение корректно созданными глобальными переменными компилируемой программы.

Это замечание сделано для того, чтобы вы поняли, что уравнения (1) и (2) касаются не только значений π и `(eval 'π)`, а и всех иных побочных эффектов вроде создания новых переменных, изменения значений старых, переходов и т. д. Именно в таком смысле надо понимать эквивалентность выражений в уравнении (2): они должны быть одинаковы во всём, везде и всегда. Должно быть абсолютно невозможным[5] составить программу, которая смогла бы различить π и `(eval 'π)` по поведению.

8.3. `eval` как функция

Специальная форма `eval` вычисляет свой единственный аргумент подобно функциям, так что возникает логичный вопрос: можно ли добиться того же поведения с помощью функции, а не специальной формы? Для ответа на него давайте снова обратимся к нашим интерпретаторам и попробуем написать для каждого из них соответствующий примитив.

С наивным интерпретатором из первой главы [**см. стр. 22**] всё просто:

```
(defprimitive eval
  (lambda (v)
    (if (program? v)
        (evaluate v env.global)
        (wrong "Illegal program" v) ) ) )
```

Сразу же в глаза бросается один серьёзный момент: `eval` недоступно текущее лексическое окружение. Так как в `evaluate` необходимо передать хоть что-то, мы берём единственное окружение, которое есть в наличии, — глобальное! Следовательно, эта функция-вычислитель работает на самом верхнем уровне вложенности — уровне глобальных определений. Назовём её `eval/at`, от *eval at top-level*, а предыдущий вариант будем называть `eval/ce`, если нам потребуется их различать. Функция `eval/at` не так уж и плоха, какой может

[5] Это утверждение носит слегка теоретический характер, так как с практической точки зрения можно засечь время выполнения обеих программ. Более медленная, очевидно, использует `eval`.

показаться на первый взгляд: ей всё же не требуется сохранять текущее лексическое окружение, что экономит немного памяти.

Для пояснения поведения `eval/at` воспользуемся предыдущим примером:

```
(set! x 2) (set! z 1)
((lambda (x) (eval/at 'x))
 3 )                          → 2
((lambda (x y) (eval/at y))
 4 'x )                       → 2
((lambda (x y z) (eval/at y))
 5 (list 'eval/at 'z) 'x )    → 1
```

При необходимости можно (приближённо) выразить `eval/at` через `eval/ce`:

```
(define (eval/at x) (eval/ce x))
```

К сожалению, в захватываемом `eval/ce` окружении оказывается одна лишняя переменная — локальная переменная `x` скрывает одноимённую глобальную. [**см. упр. 8.3**] Похоже, эту проблему можно решить, применив более тонкий подход к управлению окружениями, поэтому попробуем реализовать `eval/at` для интерпретатора байт-кодов. Новое определение снова использует функцию `compile-and-run`, но при этом просит её не сохранять адрес возврата, так как он уже был сохранён при вызове функции `eval/at`. Как и в предыдущем случае, компиляция выполняется в единственно доступном окружении `r.init`.

```
(definitial eval
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-primitive
      (lambda ()
        (if (= arity+1 (activation-frame-argument-length *val*))
            (let ((v (activation-frame-argument *val* 0)))
              (if (program? v)
                  (compile-and-run v r.init #t)
                  (signal-exception #t (list "Illegal program" v)) ) )
            (signal-exception #t
              (list "Incorrect arity" 'eval) ) ) ) ) ) )
```

Не стоит понимать ограниченность `eval/at` глобальным уровнем чересчур буквально, ведь ещё остаются динамические переменные. Совершенно не важно, реализована `eval` как специальная форма или функция, — следующее выражение всегда будет работать так, как задумано:

```
(dynamic-let (a 22)
  (eval '(dynamic a)) ) → 22
```

Изначальный контракт `eval`, выраженный уравнением (1), не подразумевал какого-либо контекста вообще. Уравнению (2), судя по рассмотренным

примерам, `eval/at` тоже не удовлетворяет. Поэтому для неё мы составим новый контракт, записанный ниже в уравнении (3); здесь переменная v не захватывается создаваемым замыканием.

$$C[\texttt{(eval/at '}\pi\texttt{)}] \equiv \texttt{(let ((}v\ \texttt{(lambda () }\pi\texttt{))) }C[(v)]) \tag{3}$$

Чаще всего `eval`, реализуемая лисп-системами, является именно `eval/at`.

8.4. Цена `eval`

Довольно сложно оценить суммарную стоимость `eval`. Предположим, разрабатываемый программный продукт является ничем иным, как `(display "Hello world!")`. Если специальная форма `eval/ce` не встречается в программе (а это можно проверить статически), то стоимость её использования нулевая! Если она всё же используется, то тянет за собой весь компилятор, так что размер приложения увеличивается примерно на порядок; скажем, где-то с 50 килобайт до 500. Если же динамический вычислитель реализован в виде функции и нельзя доказать, что эта функция не нужна, то стоимость остаётся примерно той же. Код `eval/at` из приложения можно выбросить лишь тогда, когда компилятор абсолютно уверен в том, что она никогда не будет вызвана. Язык может серьёзно помочь с доказательством этого утверждения: например, в Scheme достаточно убедиться, что глобальная переменная `eval/at` не упоминается в программе. В Common Lisp, напротив, в общем случае данное утверждение недоказуемо, так как кроме всего прочего потребуется доказать, что никто и нигде не создал строку `"eval/at"`, не превратил её в символ с помощью `find-symbol` и не получил соответствующую функцию с помощью `symbol-function`. Уже страшно? А теперь представьте, что сама `symbol-function` вызывается таким же образом

А что насчёт самой `eval`, можно ли как-то контролировать значения её аргумента? Аналогично — нет. Это вряд ли возможно выполнить одновременно легко, быстро и статически, не лишив при этом `eval` выразительности, так что придётся считать, что ей может быть передано любое значение. Следовательно, мы не можем трогать ни одну глобальную переменную вообще, так как *всё* потенциально может быть использовано и вызвано. Скомпилированное приложение должно содержать все без исключения возможности языка, что для мощных языков вроде Common Lisp означает увеличение размеров получаемого приложения ещё на один порядок.

И это ещё далеко не самое ужасное. Если не предпринять никаких мер для строгого контроля над использованием глобального окружения, как в [QP91b], где система модулей позволяет управлять доступом на чтение и модификацию глобальных переменных, то `eval` убивает на корню многие возможности для оптимизации. Рассмотрим следующий пример:

```
(define (fib n)
  (if (< n 2) (eval (read))
      (+ (fib (- n 1)) (fib (- n 2))) ) )
```

Не будем пока обращать внимание на другие возможные варианты поведения `eval`, сконцентрируемся на том, что она способна изменить значение переменной `fib`. Следовательно, при рекурсивном вызове потребуется заново извлекать адрес функции из глобальной переменной `fib`, а не просто вслепую прыгать по `GOTO`. Так как `eval` может изменить любую переменную, нельзя больше полагаться на привычные методы анализа неизменяемости глобального окружения. Одно-единственное обращение к `eval` может привести к абсолютно непредсказуемым последствиям.

По всем перечисленным причинам `eval` считается дорогим удовольствием. Тем не менее, в большом приложении (где кода на несколько мегабайт) и во время разработки, когда всё меняется и отбрасывается, `eval` может быть весьма полезной. Действительно, если понадобится динамически проводить несложные вычисления, то часто проще и эффективнее (с точки зрения количества времени, затраченного на разработку, умноженного на зарплату разработчика) использовать для этого `eval`, нежели писать и отлаживать с нуля свой собственный интерпретатор. Лисп — удивительно расширяемый язык, и наличие `eval` в стандартной библиотеке — это его серьёзное преимущество.

8.5. Интерпретируемая `eval`

Судя по этой книге, написание интерпретатора Scheme не представляет особых трудностей. Допустим, в лисп-системе нет `eval`, может ли пользователь реализовать её самостоятельно? В конце концов, существует огромный выбор интерпретаторов, пользователю достаточно скачать понравившийся (или написать самостоятельно) и включить его в свою программу, правда? Ну, и да, и нет.

Если интерпретатор пишется на Scheme, то это может только функция, так как в Scheme нельзя определять новые специальные формы. Кроме того, возникают две проблемы иного рода.

8.5.1. Взаимозаменимость представлений

Первая из упомянутых проблем касается взаимодействия между базовым языком и написанным на нём интерпретатором. Дело в том, что интерпретатор не может вводить новые типы данных, он вынужден пользоваться типами данных, которые ему предоставляет базовый язык, чтобы программы могли понимать друг друга. Если интерпретатор пользуется точечными парами, то они должны быть точечными парами базового языка; логические значения и числа должны быть одними и теми же; функции обоих языков должны соблюдать один и тот же протокол вызова. Результат любых динамических

вычислений должен быть пригоден для использования в основной программе, как в следующем примере:

```
((eval '(lambda (x) (cons x x)))
 33 )    → (33 . 33)
```

И наоборот, интерпретатор тоже должен уметь пользоваться функциями базового языка:

```
(((eval '(lambda (f)
           (lambda (x) (f x)) ))
  list )
 44 )    → (44)
```

Один из вариантов общего протокола вызова: всегда использовать функции с произвольной арностью: `(lambda values ...)`, тогда достаточно научить интерпретатор вынимать аргументы из списков. Для интерпретатора из первой главы, к примеру, это делается соответственно его природе — элементарно и наивно:

```
(define (invoke fn args)
  (if (procedure? fn)
      (apply fn args)
      (wrong "Not a function" fn) ) )

(define (make-function variables body env)
  (lambda values
    (eprogn body (extend env variables values)) ) )
```

Кроме того, необходимо позаботиться об ошибках: внешний интерпретатор должен уметь понимать и обрабатывать ошибки, возникающие во внутреннем. Этот вопрос мы обойдём стороной.

8.5.2. Глобальное окружение

Вторая проблема касается различия представлений глобального окружения. Каждый из наших интерпретаторов определяет глобальное окружение по-своему, используя макросы `definitial`, `defprimitive` и другие. Но теперь интерпретатор обязан пользоваться глобальным окружением базового языка, или хотя бы обеспечить корректную синхронизацию его состояния. Что бы там ни было, следующая программа должна выполняться правильно и без ошибок:

```
(begin (set! foo 128)
       (eval '(set! bar (+ foo foo)))
       bar )                          → 256
```

Автономные скомпилированные приложения

Компиляторы вроде Scheme→C [Bar89] или Bigloo [Ser94] работают с файлами, в которых хранится фиксированный исходный код. Получаемые таким образом автономные приложения принципиально не могут пользоваться переменными, которые не были явно упомянуты в исходном коде. Следовательно, переменные, создаваемые `eval`, будут доступны только внутри `eval`. В таких условиях возможно лишь обеспечить интерпретатору доступ к глобальным переменным внешней программы. Ответственной за это будет пара функций `global-value` и `set-global-value!`. Компилятор автоматически формирует эти функции в процессе чтения программы. [**см. упр. 7.3**] По окончании обработки они будут содержать все используемые[6] глобальные переменные. Этот список является статическим, неизменным во время работы скомпилированного приложения.

```
(define (global-value name)
  (case name
    ((car) car)
    ...
    ((foo) foo)
    (else (wrong "No such global variable" name)) ) )

(define (set-global-value! name value)
  (case name
    ((foo) (set! foo value))
    ...
    (else (wrong "No such mutable global variable" name)) ) )
```

Интерпретатор может создавать у себя внутри столько глобальных переменных, сколько ему нужно, но ни одна из них не будет видна внешней программе. Тем не менее, интерпретатор может общаться с внешней программой посредством тех переменных, которые существовали в момент компиляции.

Интерактивная система

Если же работа с лисп-системой происходит с помощью REPL, то и система, и интерпретатор могут создавать новые переменные. Приведённая ранее программа является хорошим примером такого взаимодействия: внешняя система запрашивает значение переменной, созданной внутренним интерпретатором, который запросил значение переменной, созданной системой:

```
? (begin (set! foo 128)
         (eval '(set! bar (+ foo foo)))
         2001 )
= 2001
```

[6] Конечно же, переменные вроде `car` не включаются в определение `set-global-value!` для обеспечения их неизменяемости.

```
? bar
= 256
```

Есть несколько вариантов реализации подобного сотрудничества.

Символы

Традиционное решение заключается в использовании символов. Одним из столпов семантики Лиспа является тесная взаимосвязь понятий символа и переменной: переменные записываются с помощью символов, а символы (как показано далее) могут играть роль переменных.

Символы — это составные структуры данных, имеющие как минимум одно поле: строку с собственным именем. В Scheme символы можно создавать явно с помощью функции `string->symbol`. Имена символов уникальны. Обычно это гарантируется хеш-таблицей, связывающей строки с символами. Функция `string->symbol` сначала пытается отыскать строку в данной хеш-таблице; если такая строка там есть, то она возвращает соответствующий символ, а в противном случае `string->symbol` создаёт новый символ и заносит его в таблицу на будущее. Иногда ради ускорения поиска используются различные вспомогательные скрытые поля, связывающие группы символов между собой.

Часто символы имеют также списки свойств. Обычно они реализуются в виде P-списков, но нередко в этой роли можно встретить A-списки или хеш-таблицы. От выбора внутренней структуры данных зависят потребление памяти и скорость доступа к свойствам. Некоторые лисп-системы, например Le_Lisp, даже хранят в символах специальные кеш-поля, служащие для ускорения доступа к часто используемым свойствам вроде особенностей форматирования внешнего представления форм, начинающихся с данного символа.

Раз уж любой символ имеет столько полей, то почему бы не добавить туда ещё одно для хранения значения одноимённой глобальной переменной? В случае Lisp_2, конечно, понадобятся два поля: для переменной и для функции. Ничто не ново под солнцем, именно так часто и поступают при реализации (например, [Cha80, GP88]), называя эти поля Cval и Fval. Вдобавок предоставляются функции для доступа к этим полям. В Common Lisp они носят имена `symbol-value` и `(setf symbol-value)`, а также `symbol-function` и `(setf symbol-function)`.

Такое решение привлекательно своей стопроцентной гарантией правильной работы. Каждый символ, используемый в программе, уже был создан `string->symbol`, которую вызвала `read` в процессе чтения программы. Конечно, не всякий символ используется для хранения значения глобальной переменной, но важно, что всякая глобальная переменная гарантированно имеет соответствующий символ.

Поэтому достаточно предоставить всего две функции: `global-value` и `set-global-value!`, позволяющие просматривать и изменять значения глобальных переменных, ссылаясь на них по именам. Интерпретатор `eval` сможет воспользоваться глобальными переменными базового языка посредством этих

функций, а проблема корректного расширения глобального окружения пропадает в принципе — она автомагически решается механизмом доступа к символам.

Покажем на примере интерпретатора из первой главы, как использовать эти функции. Окружение `env` теперь будет содержать только локальные переменные, а глобальные будут использоваться напрямую — теперь они совпадают с глобальными переменными того Scheme, на котором написан интерпретатор. Таким образом, получаем:

```
(define (lookup id env)
  (if (pair? env)
      (if (eq? id (caar env))
          (cdar env)
          (lookup id (cdr env)) )
      (global-value id) ) )

(define (update! id env value)
  (if (pair? env)
      (if (eq? id (caar env))
          (begin (set-cdr! (car env) value)
                 value )
          (update! id (cdr env) value) )
      (set-global-value! id value) ) )
```

Данное решение кажется элегантным, но эта элегантность приобретается за внушительную цену: ведь теперь *любая* глобальная переменная доступна по имени. В случае автономного приложения `global-value` вынуждает нас включать в приложение всю без исключения библиотеку функций, так как любое имя может быть вычислено, а значит, любая функция может понадобиться. Это не особо критично для среды разработки, так как она затем и существует, чтобы абсолютно всё было под рукой, но вот с маленькими автономными приложениями такой подход сочетается плохо. Более того, функция `set-global-value!` может изменить значение любой изменяемой глобальной переменной, что, конечно же, оказывает закономерный эффект на оптимизации компилятора.

Пару функций `global-value` и `set-global-value!` можно понимать как специализированную версию `eval` для переменных. Между прочим, в некоторых старых лисп-системах так оно и было: этот специализированный вычислитель назывался `symeval`. Также можно их считать парой рефлексивных функций, реифицирующих доступ к глобальным переменным. Написать «`foo`», чтобы получить значение глобальной переменной `foo`, — это то же самое, что неявно вычислить значение формы `(global 'foo)`.

Полноценные окружения

Другой вариант решения проблемы окружений состоит в определении `eval` как функции двух переменных: выражения, которое необходимо вычислить, и окружения, в котором будут проводиться вычисления. Бинарная `eval` наивного интерпретатора вроде как подходит по сигнатуре, но при этом упускается один важный момент: аргументом функции может быть только значение. Внутри интерпретатора окружения просто существуют, не являясь при этом полноценными значениями языка. Следовательно, для реализации этой идеи необходимо определить операцию реификации окружений, превращающую их в полноценные объекты. Кроме того, было бы полезным иметь и другие возможности вроде модификации окружений или хотя бы извлечения значений переменных из них. Наличие соответствующих функций позволяет реализовать механизм взаимодействия окружений интерпретатора с окружениями внешней системы. Такой подход поднимает множество специфичных вопросов, которые мы сейчас и рассмотрим.

8.6. Реификация окружений

Денотационный интерпретатор чётко показал, что контекст вычислений в Scheme является тройкой из окружения, продолжения и памяти. Отложив память в дальний угол, сфокусируемся на оставшейся паре. Продолжения уже могут становиться полноценными объектами языка благодаря `call/cc` или `bind-exit`. Окружениями же возможно пользоваться только неявно. В данном разделе мы рассмотрим, как превратить их в объекты, допускающие явную программную манипуляцию.

Добиться этого можно различными способами, сохраняющими те или иные свойства, допускающими те или иные операции. Подробную классификацию можно найти в [RA82, MR91]. Существуют три фундаментальные операции над окружениями: поиск значения переменной, изменение значения переменной и добавление новой переменной в окружение. Реификация окружений фактически сводится к захвату соответствующих привязок переменных к значениям. Привязки при этом становятся явными, что позволяет реализовать вышеприведённые операции как обычные функции языка.

8.6.1. Специальная форма `export`

Начнём со специальной формы, называемой `export`: ей передаётся список имён переменных, а в ответ она возвращает окружение с соответствующими привязками. Полученное окружение можно будет впоследствии передать вторым аргументом в бинарную функцию `eval` (впредь будем называть её `eval/b`). Например:

```
(let ((r (let ((x 3)) (export x))))
  (eval/b 'x r) )                    → 3
```

```
(let ((f+r (let ((x 3))
             (cons (lambda () x) (export x)) )))
  (eval/b '(set! x (+ 1 x)) (cdr f+r))
  ((car f+r)) )                          → 4

(let ((r (export car cons)))
  (let ((car cdr))
    (eval/b '(car (cons 1 2)) r) ) ) → 1
```

Первый пример показывает, что объект-окружение, содержащий привязку переменной `x`, действительно создаётся, а `eval/b` это окружение успешно переваривает. Второй пример демонстрирует изменяемость переменной `x` и доказывает, что захватывается действительно привязка, так как иным способом изменить значение свободной переменной замыкания невозможно. Третий пример показывает, что можно захватывать и глобальные привязки.

Специальная форма `export` позволяет жонглировать окружениями по желанию: захватить окружение в одном месте, а использовать в совершенно другом. Реализация данной формы нетривиальна, так что её стоит рассмотреть подробнее. Как обычно, начинаем с добавления новой формы в синтаксический анализатор `meaning`:

```
... ((export) (meaning-export (cdr e) r tail?)) ...
```

Естественно, захваченное окружение должно содержать не только список записей активаций, но также полный перечень имён и адресов переменных. Та же информация была нужна `eval/ce`, как вы помните. Поэтому реифицированные окружения будут представляться объектами с двумя полями: списком записей активаций и списком пар «имя — адрес», по которому программы смогут разобраться, что где находится. (См. рисунок 8.2.)

```
(define-class reified-environment Object
  ( sr r ) )
```

Не особо (пока) углубляясь в детали, будем компилировать форму `export` следующим образом:

```
(define (meaning-export n* r tail?)
  (unless (every? symbol? n*)
          (static-wrong "Incorrect variables" n*) )
  (append (CONSTANT (extract-addresses n* r)) (CREATE-1ST-CLASS-ENV)) )

(define (CREATE-1ST-CLASS-ENV) (list 254))

(define-instruction (CREATE-1ST-CLASS-ENV) 254
  (create-first-class-environment *val* *env*) )

(define (create-first-class-environment r sr)
  (set! *val* (make-reified-environment sr r)) )
```

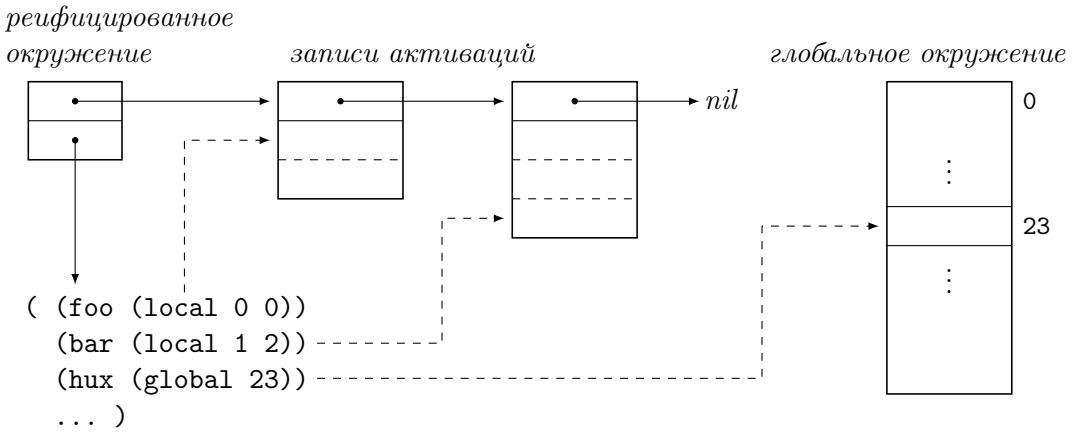


Рис. 8.2. Представление окружений.

Список пар «имя — адрес» передаётся как цитата через регистр `*val*`, его оттуда забирает новая инструкция `CREATE-1ST-CLASS-ENV`, которая создаст соответствующий объект.

Для упрощения вычислений мы немного изменим представление статической части окружений (значений переменной `r`). Вместо «гирлянд» они теперь будут просто ассоциативными списками, как на рисунке 8.2. Это небольшое изменение, но упрощение `compute-kind` приобретается ценой усложнения `r-extend*`: теперь при каждом расширении окружения необходимо переносить уже существующие переменные на один уровень глубже.

```
(define (compute-kind r n)
  (or (let ((var (assq n r)))
        (and (pair? var) (cadr var)) )
      (global-variable? g.current n)
      (global-variable? g.init n)
      (adjoin-global-variable! n) ) )

(define (r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*) (i 0))
      (cond ((pair? n*) (cons (list (car n*) `(local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r)
            (else (cons (list n* `(local 0 . ,i)) old-r)) ) ) ) )

(define (bury-r r offset)
  (map (lambda (d)
         (let ((name (car d))
               (type (car (cadr d))) )
```

```
            (case type
              ((local checked-local)
               (let* ((addr (cadr d))
                      (i (cadr addr))
                      (j (cddr addr)) )
                 `(,name (,type ,(+ i offset) . ,j) . ,(cddr d)) ) )
              (else d) ) ) )
        r ) )
```

Помимо возможности создания окружения исключительно из упомянутых переменных, мы также реализуем возможность захвата всего окружения целиком: форма `(export)` будет эквивалентна `(export` *переменные*`...)`, где перечислены все локальные переменные, созданные окружающими форму абстракциями. Имея форму `(export)`, часто именуемую `(the-environment)`, можно проэмулировать `eval/ce` с помощью `eval/b`:

$$\texttt{(eval/ce '}\pi\texttt{)} \equiv \texttt{(eval/b '}\pi\texttt{ (export))}$$

Реализовать эту возможность проще простого, так как теперь окружения изначально имеют необходимую структуру.

```
(define (extract-addresses n* r)
  (if (null? n*) r
      (let scan ((n* n*))
        (if (pair? n*)
            (cons (list (car n*) (compute-kind r (car n*)))
                  (scan (cdr n*)) )
            '() ) ) ) )
```

8.6.2. Функция `eval/b`

На этом сходства `eval/b` с `eval/ce` не заканчиваются. Они похожи ещё в том, как получают параметры и возвращают значения. Будет полезным сравнить рассматриваемую далее реализацию `eval` с предыдущими. Функция `eval/b` проверяет аргументы на корректность, после чего передаёт работу знакомой вам функции `compile-on-the-fly`, чтобы она скомпилировала выражение в окружении, разместила результат в памяти и передала ему управление. Текущее окружение сохранять не требуется, так как за это отвечает протокол вызова функций.

```
(definitial eval/b
  (let* ((arity 2)
         (arity+1 (+ 1 arity)) )
    (make-primitive
      (lambda ()
        (if (= arity+1 (activation-frame-argument-length *val*))
            (let ((exp (activation-frame-argument *val* 0))
                  (env (activation-frame-argument *val* 1)) )
```

```
                (if (program? exp)
                    (if (reified-environment? env)
                        (compile-and-evaluate exp env)
                        (signal-exception
                         #t (list "Not an environment" env) ) )
                    (signal-exception
                     #t (list "Illegal program" exp) ) ) )
               (signal-exception
                #t (list "Incorrect arity" 'eval/b) ) ) ) ) ) )

(define (compile-and-evaluate v env)
  (let ((r (reified-environment-r env))
        (sr (reified-environment-sr env)) )
    (set! *env* sr)
    (set! *pc* (compile-on-the-fly v r)) ) )
```

8.6.3. Расширяем окружения

Так как окружения расширяются на каждом шагу, то будет нелишним дать такую возможность и пользователям языка. Для этого мы определим функцию `enrich`, принимающую окружение и список добавляемых имён. Возвращаемым значением будет *новое* расширенное окружение. `enrich` — это чисто функциональный модификатор, она не изменяет свои аргументы. Рассмотрим пример её использования — ручную эмуляцию `letrec`. Сначала захватывается глобальная привязка,[7] затем к ней добавляются привязки двух локальных переменных `odd?` и `even?`, после чего определяется пара взаимно рекурсивных функций и, наконец, выполняются вычисления с их помощью.

```
((lambda (e)
   (set! e (enrich (export *) 'even? 'odd?))
   (eval/b '(set! even? (lambda (n) (if (= n 0) #t (odd? (- n 1))))) e)
   (eval/b '(set! odd? (lambda (n) (if (= n 0) #f (even? (- n 1))))) e)
   (eval/b '(even? 4) e) )
 'ee )                          → #t
```

Структура используемых окружений показана на рисунке 8.3. Вначале создаётся запись активации для хранения новых переменных, затем создаётся окружение, связывающее имена переменных с новыми адресами. Единственная проблема состоит в том, что новые переменные объективно существуют, но ещё не имеют значений. С аналогичными затруднениями при реализации формы `letrec` мы уже сталкивались ранее. [**см. стр. 83**]

[7] А вот и архитектурная ошибочка нашлась: `export` не позволяет захватить *пустое* окружение! Поэтому мы захватываем хоть что-нибудь — в данном случае функцию умножения, — чтобы получить начальное окружение для манипуляций (аналогичный костыль применён в [QD96]).

Решением будет новый тип адресов: `checked-local`, являющийся локальным аналогом `checked-global`. Текущее определение `enrich` допускает существование локальных переменных, которые не имеют значений, что было невозможным при использовании `lambda`. Поэтому нам потребуется специальная разновидность привязок для данного случая. Конечно, как вариант можно было бы разрешить расширять окружения исключительно инициализированными переменными.

Написать определение `enrich` теперь не составит большого труда, пусть оно и получилось весьма объёмным:

```
(definitial enrich
  (let ((arity+1 (+ 1 1)))
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
         (let ((env (activation-frame-argument *val* 0)))
           (listify! *val* 1)
           (if (reified-environment? env)
             (let* ((names (activation-frame-argument *val* 1))
                    (len (- (activation-frame-argument-length *val*)
                            2 ))
                    (r (reified-environment-r env))
                    (sr (reified-environment-sr env))
                    (frame (allocate-activation-frame
                            (length names) )) )
               (set-activation-frame-next! frame sr)
               (do ((i (- len 1) (- i 1)))
                   ((< i 0))
                 (set-activation-frame-argument! frame i
                                                 undefined-value ) )
               (unless (every? symbol? names)
                 (signal-exception
                  #f (list "Incorrect variable names" names) ) )
               (set! *val* (make-reified-environment frame
                            (checked-r-extend* r names) ))
               (set! *pc* (stack-pop)) )
             (signal-exception
              #t (list "Not an environment" env) ) ) )
         (signal-exception
          #t (list "Incorrect arity" 'enrich) ) ) ) ) ) )

(define (checked-r-extend* r n*)
  (let ((old-r (bury-r r 1)))
    (let scan ((n* n*) (i 0))
      (cond ((pair? n*) (cons (list (car n*) `(checked-local 0 . ,i))
                              (scan (cdr n*) (+ i 1)) ))
            ((null? n*) old-r) ) ) ) )
```

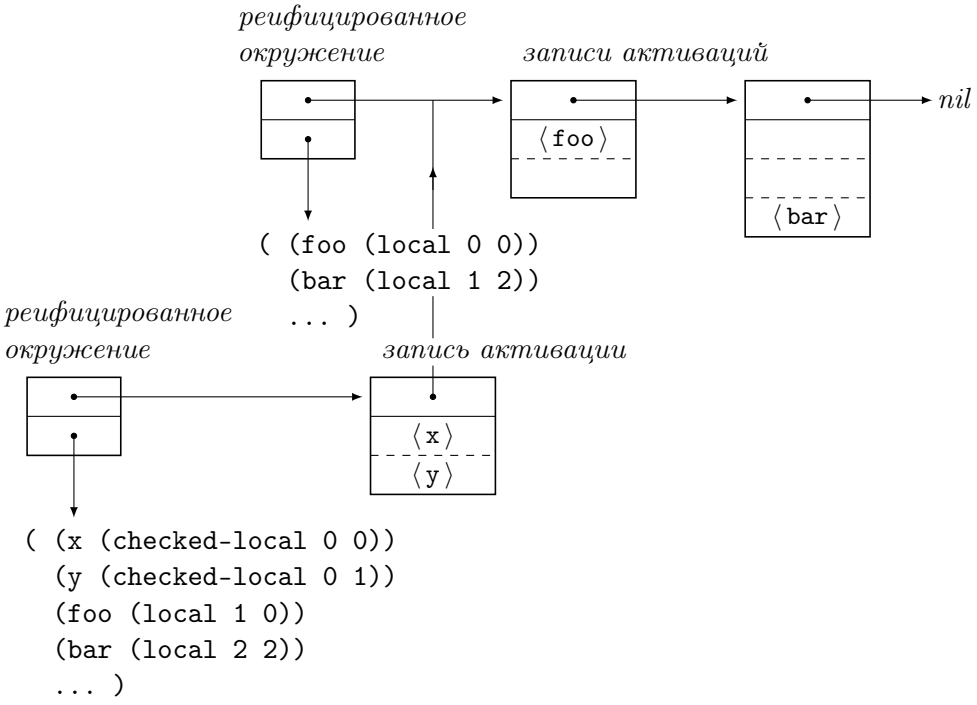


Рис. 8.3. Расширение окружения: `(enrich` *env* `'x 'y)`.

Также необходимо дополнить определения `meaning-reference` и `meaning-assignment`, дабы они учитывали новый тип привязок (`checked-local`).

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind (case (car kind)
               ((checked-local)
                (let ((i (cadr kind))
                      (j (cddr kind)) )
                  (CHECKED-DEEP-REF i j) ) )
               ((local)
                (let ((i (cadr kind))
                      (j (cddr kind)) )
                  (if (= i 0)
                      (SHALLOW-ARGUMENT-REF j)
                      (DEEP-ARGUMENT-REF i j) ) ) )
               ((global)
                (let ((i (cdr kind)))
                  (CHECKED-GLOBAL-REF i) ) )
               ((predefined)
                (let ((i (cdr kind)))
                  (PREDEFINED i) ) ) )
        (static-wrong "No such variable" n) ) ) )
```

```
(define (meaning-assignment n e r tail?)
  (let ((m (meaning e r #f))
        (kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((local checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-SET! j m)
                 (DEEP-ARGUMENT-SET! i j m) ) ) )
          ((global)
           (let ((i (cdr kind)))
             (GLOBAL-SET! i m) ) )
          ((predefined)
           (static-wrong "Immutable predefined variable" n) ) )
        (static-wrong "No such variable" n) ) ) )
```

И, конечно же, не забываем о новой инструкции `CHECKED-DEEP-REF`:

```
(define-instruction (CHECKED-DEEP-REF i j) 253
  (set! *val* (deep-fetch *env* i j))
  (when (eq? *val* undefined-value)
    (signal-exception #t (list "Uninitialized local variable")) ) )
```

8.6.4. Анатомия замыканий

Некоторые интерпретаторы предоставляют примитивы, позволяющие извлечь из замыкания хранимое в нём окружение. Кроме функции `procedure->environment` также часто имеется и `procedure->definition`, которая возвращает код замыкания. Эту пару функций довольно легко реализовать для интерпретатора, но в случае компилятора задача уже не такая тривиальная и требует несколько большего объёма памяти. Им необходимо сохранить определения функций (которые занимают место в памяти), а также явно представить (в памяти) структуру замкнутых окружений. Более того, функция `procedure->environment` пагубно влияет на оптимизацию, так как теперь становится возможным добраться даже до святая святых — замкнутых свободных переменных, прежде недоступных вообще никому, кроме собственно замыканий. Функция `procedure->environment` в этом плане ещё хуже, чем `export`: последняя хотя бы ограничивает негативные последствия явно указанными переменными.

Функция `procedure->definition` полезна своими интроспективными возможностями. С её помощью можно создать отладчик, обладающий полным контролем над отлаживаемыми программами. Например, рассмотрим функцию `trace-procedure1`, которая принимает унарную функцию, разбирает её

на части и собирает из них новую унарную функцию, аналогичную исходной, но выводящую на экран свой аргумент и результат выполнения.

```
(define (trace-procedure1 f)
  (let* ((env (procedure->environment f))
         (definition (procedure->definition f))
         (variable (car (car definition)))
         (body (cddr definition)) )
    (lambda (variable)
      (display (list 'entering f 'with value))
      (eval/b `(begin (set! ,variable ',value)
                      (let ((result (begin . ,body)))
                        (display (list 'result 'is result))
                        result ) )
              (enrich env variable) ) ) ) )
```

Если быть честным, эта программа немного жульничает. Встраиваемая функция вроде `car` её сломает, а код, синтезируемый для `eval/b`, не обязательно является корректным, так как цитирует значение переменной `value`, которое может оказаться нецитируемым (к примеру, замыканием). Тем не менее, `trace-procedure1` является хорошим примером взаимодействия интроспекции, полноценных окружений и динамических вычислений.

Давайте рассмотрим пример реализации данных функций, чтобы вы лучше представляли себе их стоимость. Функция `procedure->definition` более проста: ей надо всего лишь связать замыкание с его определением, то есть с цитатой соответствующей абстракции. Проблема состоит в том, куда поместить эту цитату, так как несколько замыканий могут иметь одинаковое определение. Вспомним, что замыкания ассоциируются с кодом, хранимым по определённому адресу. Все замыкания данной абстракции ссылаются на один и тот же код, так что цитируемое определение разумно разместить где-то поблизости. Подобное перемежение кода и данных хорошо известно программистам на ассемблере. Иллюстрация идеи приведена на рисунке 8.4.

Реализация данной идеи тривиальна.[8] Для нужд `procedure->environment` и `procedure->definition` в кодогенераторы абстракций необходимо передавать определения соответствующих функций и их окружения. n-арный вариант данной функции легко написать по аналогии:

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (r2 (r-extend* r n*))
         (m+ (meaning-sequence e+ r2 #t)) )
    (REFLECTIVE-FIX-CLOSURE m+ arity
      `(lambda ,n* . ,e+) r ) ) )
```

[8] `EXPLICIT-CONSTANT` используется для того, чтобы не получить `(PREDEFINED0)` вместо ожидаемого `(CONSTANT '())`.

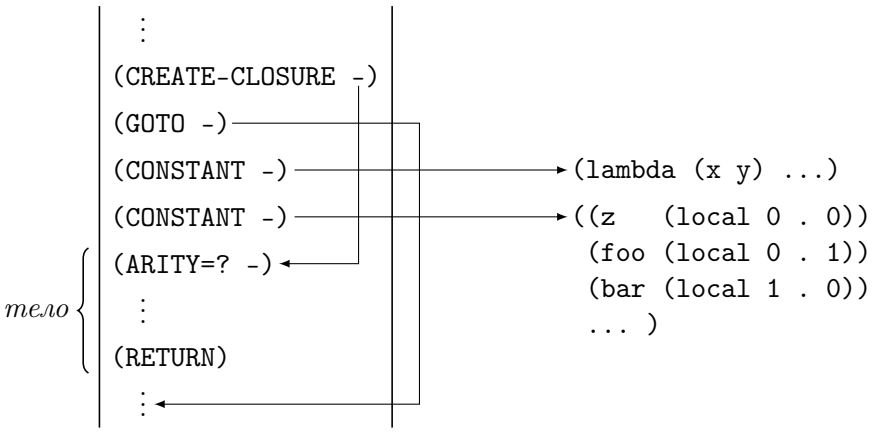


Рис. 8.4. Рефлексивная абстракция.

```
(define (REFLECTIVE-FIX-CLOSURE m+ arity definition r)
  (let* ((the-function (append (ARITY=? (+ arity 1)) (EXTEND-ENV)
                               m+ (RETURN) ))
         (the-env (append (EXPLICIT-CONSTANT definition)
                          (EXPLICIT-CONSTANT r) ))
         (the-goto (GOTO (+ (length the-env) (length the-function)))) )
    (append (CREATE-CLOSURE (+ (length the-goto) (length the-env)))
            the-goto the-env the-function ) ) )
```

Функция `procedure->definition` извлекает определение замыкания, расположенное на две инструкции позади его кода.

```
(definitial procedure->definition
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let ((pc (closure-code proc)))
                   (set! *val* (vector-ref *constants*
                                           (vector-ref *code* (- pc 2)) ))
                   (set! *pc* (stack-pop)) )
                 (signal-exception #f (list "Not a procedure" proc)) ) )
           (signal-exception #t
            (list "Incorrect arity" 'procedure->definition) ) ) ) ) ) )
```

Функция `procedure->environment` извлекает замкнутое окружение и реифицирует его, устанавливая в нём соответствия между именами переменных

и их адресами. Таблица символов хранится на одну инструкцию позади тела замыкания.

```
(definitial procedure->environment
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-primitive
     (lambda ()
       (if (>= (activation-frame-argument-length *val*) arity+1)
           (let ((proc (activation-frame-argument *val* 0)))
             (if (closure? proc)
                 (let* ((pc (closure-code proc))
                        (r (vector-ref *constants*
                               (vector-ref *code* (- pc 1)) )) )
                   (set! *val* (make-reified-environment
                                (closure-closed-environment proc)
                                r ))
                   (set! *pc* (stack-pop)) )
                 (signal-exception #f (list "Not a procedure" proc)) ) )
           (signal-exception #t
            (list "Incorrect arity" 'procedure->environment) ) ) ) ) ) )
```

Вкратце, `procedure->definition` и `procedure->environment` позволяют программам осознавать свой собственный код. Также их возможности полезны для написания интроспективных отладочных инструментов. Однако, нельзя забывать о довольно высокой стоимости использования данных функций ввиду существенного объёма сохраняемой ими информации. Более того, они отличаются полным отсутствием такта: сохраняют и выставляют напоказ всё, что видят. Никакого проприетарного кода, никакого сокрытия реализаций структур данных, и никаких локальных оптимизаций, так как в подобных условиях неизменяемость и недоступность чего-либо не гарантируется.

Часть недостатков этих функций исправляется введением дополнительной специальной формы, скажем, `reflective-lambda`, имеющей следующий синтаксис:

`(reflective-lambda (`*переменные*`) (`*экспортируемые переменные*`)`
  тело `)`

Подобно `lambda`-абстракциям, которые она обобщает, первым аргументом ей передаются локальные переменные, а в конце идёт тело функции, но между ними расположен список экспортируемых переменных. Лишь перечисленные свободные переменные будут видны для внешнего мира, если извлечь сохранённое окружение из подобного замыкания. Обычной абстракции `(lambda (`*переменные*`)` *тело*`)` соответствует новая `(reflective-lambda (`*переменные*`) () `*тело*`)`, которая не экспортирует ничего. Такая специальная форма позволяет контролировать доступ к привязкам, скрывая при необходимости то, что должно быть личным.

Наконец, остаётся ещё один щекотливый вопрос, поднимаемый чрезмерно любопытной функцией `procedure->environment`. Какое именно окружение замыкает в себе следующая абстракция?

```
(let ((x 1) (y 2))
  (lambda (z) x) )
```

Это окружение однозначно содержит `x`, так как она является свободной в теле абстракции. Но захватывается ли `y`, которая тоже присутствует в лексическом окружении? Для рассмотренной реализации это так, потому что функция `procedure->environment` вынуждена соблюдать контракт, который подразумевает, что с помощью полученного окружения возможно вычислить любое выражение (в том числе использующее переменную `y`), как будто бы оно вычислялось внутри соответствующего замыкания. Поэтому `procedure->environment` и возвращает целиком и полностью окружение, используемое при создании замыкания.

8.6.5. Специальная форма `import`

Часто форма, передаваемая динамическому вычислителю, является по сути статической. Например, такая ситуация наблюдается в функции `trace-procedure1`. Вместо полной компиляции на лету здесь бы отлично подошла предкомпиляция вычисляемой формы, оставляющая динамическим только выбор окружения, в котором будут проходить вычисления. Введём для этих целей новую специальную форму `import` со следующим синтаксисом:

`(import (`*переменные*`...) `*окружение*
  формы`... )`

Формы, составляющие тело `import`, вычисляются в специальном окружении. Свободные *переменные*, присутствующие[9] в переданном списке, берутся из *окружения*, а остальные, как обычно, извлекаются из лексического окружения самой формы `import`. Список заменяемых переменных статический, их имена не могут быть вычислены; сначала вычисляется *окружение*, затем *формы*.

Рассмотрим пример её использования, вдохновлённый Meroonet. Там возникли затруднения с представлением обобщённых функций, которые одновременно должны быть и функциями, и объектами. Если разрешить доступ к замкнутым свободным переменным извне, то замыкания можно считать объектами, полями которых являются свободные переменные. На этой идее основывается[10] отождествление замыканий с объектами. В Meroonet для добавления метода обобщённой функции его необходимо разместить по определённому индексу в векторе методов. Если обобщённая функция замыкает

[9] Пустой список, подобно `export`, может означать все возможные переменные.

[10] Помимо этого необходимы ещё многие другие возможности, вроде механизма диспетчеризации методов по классам объектов.

в себе этот вектор под именем `methods`, то данное действие можно записать прямо:

```
(define (add-method! generic class method)
  (import (methods) (procedure->environment generic)
    (vector-set! methods (Class-number class) method) ) )
```

Здесь отлично видно преимущество `import` над `eval/b`. Переменные `class` и `method` внутри функции известны статически, поэтому компилятор не может сразу же подставить лишь адрес переменной `methods` — он станет известным только после вычисления второго аргумента `import`. В итоге мы получаем динамическую компиляцию наподобие `eval/b`, но выполняем статически большую часть работы: собственно синтез исполнимого кода. Фактически, форме `import` остаётся только связать определённые переменные с вычисленными значениями. Подобное связывание в [LF93] и [NQ89] называется *квазистатическим*. Помимо этого, само слово `import` намекает на тесную связь с `export`: одна форма производит окружения, которыми пользуется другая. С их помощью уже можно соорудить простую, но полноценную (в обоих смыслах) систему модулей.

Итак, как же реализуется специальная форма `import`? Прежде всего, конечно, в синтаксическом анализаторе `meaning` появляется новая строка:

```
...
((import) (meaning-import (cadr e) (caddr e) (cdddr e) r tail?)) ...
```

Генерируемый байт-код сохраняет список несвязанных переменных (назовём их *плавающими*) в стеке, вычисляет переданное окружение и сохраняет результат в `*val*`, после чего выполняет новую инструкцию `CREATE-PSEUDO-ENV` и передаёт управление телу формы `import`, принимая во внимание `tail?`.

```
(define (meaning-import n* e e+ r tail?)
  (let* ((m (meaning e r #f))
         (r2 (shadow-extend* r n*))
         (m+ (meaning-sequence e+ r2 #f)) )
    (append (CONSTANT n*) (PUSH-VALUE)
            m (CREATE-PSEUDO-ENV)
            (if tail? m+ (append m+ (UNLINK-ENV))) ) ) )
```

Перед вычислением тела специальной формы `import` текущее окружение расширяется псевдозаписью активации (объектом класса `pseudo-activation-frame`), хранящей информацию о плавающих переменных. Она может свободно связываться в цепочку с другими записями активаций, но внутри себя содержит не значения переменных, а их адреса вместе с окружением, относительно которого они вычислены. При создании псевдозаписи активации для выяснения адресов переменных используется функция `compute-kind`.

```
(define-instruction (CREATE-PSEUDO-ENV) 252
  (create-pseudo-environment (stack-pop) *val* *env*) )
```

```
(define-class pseudo-activation-frame environment
  ( sr (* address) ) )

(define (create-pseudo-environment n* env sr)
  (unless (reified-environment? env)
    (signal-exception #f (list "Not an environment" env)) )
  (let* ((len (length n*))
         (frame (allocate-pseudo-activation-frame len)) )
    (let setup ((n* n*) (i 0))
      (when (pair? n*)
        (set-pseudo-activation-frame-address!
         frame i (compute-kind (reified-environment-r env) (car n*)) )
        (setup (cdr n*) (+ i 1)) ) )
    (set-pseudo-activation-frame-sr! frame
     (reified-environment-sr env) )
    (set-pseudo-activation-frame-next! frame sr)
    (set! *env* frame) ) )
```

Тело специальной формы `import` следует компилировать с учётом новых плавающих переменных. Они помещаются в окружение компиляции `r` с помощью функции `shadow-extend*`. Как известно, есть только одна проблема, которую нельзя решить введением дополнительного уровня косвенности. Поэтому значения плавающих переменных определяются с помощью косвенной адресации: сначала из псевдозаписи активации извлекается адрес, а затем в соответствующем окружении ищется необходимое значение. На рисунке 8.5 приведена иллюстрация структуры окружений для следующей программы:

```
(let ((x 11))
  (let ((z 22)
        (env (let ((z 33)) (export))) )
    (import (x y) env
      (list (set! x y) z) ) ) )
```

Плавающие переменные должны компилироваться по-своему, так что они помечаются для компилятора специальной функцией `shadow-extend*`:

```
(define (shadow-extend* r n*)
  (let enum ((n* n*) (j 0))
    (if (pair? n*)
        (cons (list (car n*) `(shadowable 0 . ,j))
              (enum (cdr n*) (+ j 1)) )
        (bury-r r 1) ) ) )
```

Также необходимо дополнить определения `meaning-reference` и `meaning-assignment` предобработкой нового типа переменных. Помимо этого ещё потребуется ввести две новые инструкции: `SHADOW-REF` и `SHADOW-SET!`, отвечающие за взаимодействие с подобными переменными на машинном уровне.

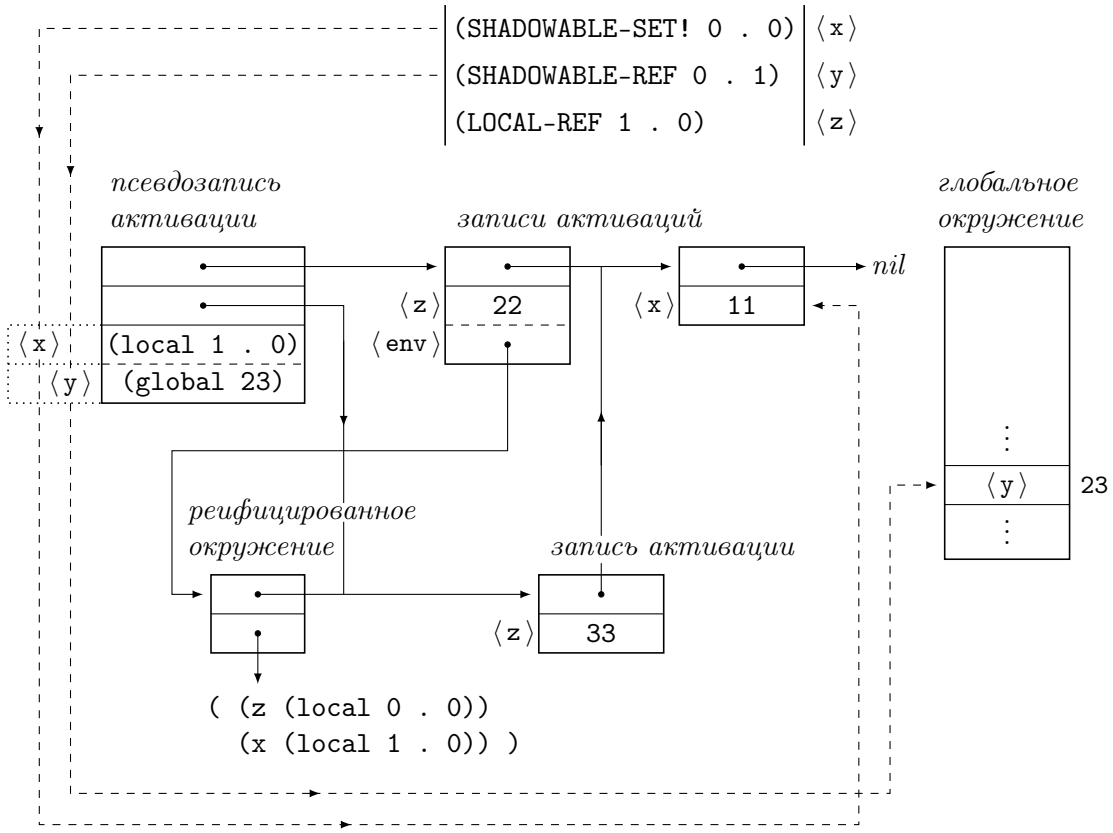


Рис. 8.5. Принцип работы `import`.

```
(define (meaning-reference n r tail?)
  (let ((kind (compute-kind r n)))
    (if kind
        (case (car kind)
          ((checked-local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (CHECKED-DEEP-REF i j) ) )
          ((local)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (if (= i 0)
                 (SHALLOW-ARGUMENT-REF j)
                 (DEEP-ARGUMENT-REF i j) ) ) )
          ((shadowable)
           (let ((i (cadr kind))
                 (j (cddr kind)) )
             (SHADOW-REF i j) ) )
```

```
           ((global)
            (let ((i (cdr kind)))
              (CHECKED-GLOBAL-REF i) ) )
           ((predefined)
            (let ((i (cdr kind)))
              (PREDEFINED i) ) ) )
         (static-wrong "No such variable" n) ) ) )

(define-instruction (SHADOW-REF i j) 231
  (shadowable-fetch *env* i j) )

(define (shadowable-fetch sr i j)
  (if (= i 0)
      (let ((kind (pseudo-activation-frame-address sr j))
            (sr (pseudo-activation-frame-sr sr)) )
        (variable-value-lookup kind sr) )
      (shadowable-fetch (environment-next sr) (- i 1) j) ) )

(define (variable-value-lookup kind sr)
  (if (pair? kind)
      (case (car kind)
        ((checked-local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (set! *val* (deep-fetch sr i j))
           (when (eq? *val* undefined-value)
             (signal-exception #t
               (list "Uninitialized local variable") ) ) ) )
        ((local)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (set! *val*
            (if (= i 0)
                (activation-frame-argument sr j)
                (deep-fetch sr i j) ) ) ) )
        ((shadowable)
         (let ((i (cadr kind))
               (j (cddr kind)) )
           (shadowable-fetch sr i j) ) )
        ((global)
         (let ((i (cdr kind)))
           (set! *val* (global-fetch i))
           (when (eq? *val* undefined-value)
             (signal-exception #t
               (list "Uninitialized global variable") ) ) ) )
```

```
      ((predefined)
       (let ((i (cdr kind)))
         (set! *val* (predefined-fetch i)) ) ) )
    (signal-exception #f (list "No such variable")) ) )
```

Функция `shadowable-fetch` использует статически известный промежуточный адрес, чтобы получить динамический адрес, по которому фактически располагается искомая переменная. При этом необходимо различать все возможные типы фактических адресов, а именно: `local` и `checked-local`, `global` и `predefined`, и даже `shadowable`, ведь это вполне возможный вариант. Функция `variable-value-lookup` выполняет эту работу. В некотором смысле `import` можно понимать как макрос, преобразующий ссылки на плавающие переменные в вызовы `eval/b`. Например, форма `(import (x y) env (list (set! x y) z))` из предыдущего примера эквивалентна[11] следующей:

```
(list ((eval/b '(lambda (v) (set! x v)) env)
       (eval/b 'y env) )
      z )
```

Подведём некоторый итог, ещё раз взглянув на все типы привязок, с которыми мы встречались.

Форма `lambda` создаёт лексические привязки — коробки для хранения значений переменных. Коробки имеют имена, позволяющие ссылаться на них. Время жизни коробок не ограничено: они исчезают лишь тогда, когда никто больше не сможет ими воспользоваться. Область видимости имён коробок ограничивается телом `lambda`-формы, которая их создала.

Форма `dynamic-let` создаёт динамические привязки — связи имён со значениями,[12] устанавливая их на время определённых вычислений. Область видимости данных привязок не ограничена, однако они перестают существовать с окончанием соответствующих вычислений.

Квазистатические привязки являются альтернативными именами уже существующих привязок. Подобная множественность имён несовместима с α-конверсией.* Почему? Потому что даже ограниченные способы[13] реификации окружений, не такие всеобъемлющие, как `procedure->environment`, дают возможность пользоваться привязками в совершенно произвольных местах, без учёта их нормальной области видимости и изначального имени.

[11] Чтобы избежать потенциально ошибочного цитирования присваиваемого значения, используется промежуточное замыкание, принимающее это значение.

[12] Рассмотренная реализация `dynamic-let` не позволяет напрямую изменять такие связи; не существует соответствующей `dynamic-set!`. Но ничто не мешает связать с именем изменяемый объект и таким образом обойти данное ограничение.

* В λ-исчислении α-конверсией называется правило $\lambda x.N \equiv \lambda y.N[x \to y]$, которое даёт возможность свободно переименовывать связанные переменные абстракции при условии, что новое имя y не встречается в N.

[13] Кстати, можно улучшить `reflective-lambda`, дав ей возможность указывать операции, допустимые над экспортируемыми привязками; например, разрешать только чтение, но не изменение значений. Также можно, подобно [LF93], реализовать механизм переименования экспортируемых переменных.

8.6.6. Упрощённый доступ к окружениям

Предыдущие разделы показали, что реифицированные окружения отлично сочетаются с явными вычислениями. Единственным исключением, которое пока не удалось выразить с их помощью, остаются программы, использующие глобальное окружение. Раз уж сами по себе явные объекты-окружения не способны унифицировать обращения к переменным, то попробуем зайти с другой стороны: обобщим функции `global-value` и `set-global-value!` на случай любых переменных, получая `variable-value`, `set-variable-value!` и `variable-defined?`.

Функция `variable-value` используется для поиска значения переменной в переданном ей окружении; `set-variable-value!` изменяет данное значение; `variable-defined?` проверяет наличие переменной в окружении. Все они извлекают фактический адрес искомой переменной аналогично `shadowable-fetch`: в обход функции `compute-kind`, так как она создаёт глобальные переменные на лету, если не может их найти в локальном окружении. Определение `set-variable-value!` не приводится для краткости, но его довольно легко восстановить по имеющимся.

```
(definitial variable-value
  (let* ((arity 2)
         (arity+1 (+ 1 arity)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
                            (sr (reified-environment-sr env))
                            (kind
                             (or (let ((var (assq name r)))
                                   (and (pair? var) (cadr var)) )
                                 (global-variable? g.current name)
                                 (global-variable? g.init name) ) ) )
                       (variable-value-lookup kind sr)
                       (set! *pc* (stack-pop)) )
                     (signal-exception
                      #f (list "Not a variable name" name) ) )
                 (signal-exception
                  #t (list "Not an environment" env) ) ) )
           (signal-exception
            #t (list "Incorrect arity" 'variable-value) ) ) ) ) ) )
```

```
(definitial variable-defined?
  (let* ((arity 2)
         (arity+1 (+ 1 arity)) )
    (make-primitive
     (lambda ()
       (if (= (activation-frame-argument-length *val*) arity+1)
           (let ((name (activation-frame-argument *val* 0))
                 (env (activation-frame-argument *val* 1)) )
             (if (reified-environment? env)
                 (if (symbol? name)
                     (let* ((r (reified-environment-r env))
                            (sr (reified-environment-sr env)) )
                       (set! *val*
                             (if (or (let ((var (assq name r)))
                                       (and (pair? var) (cadr var)) )
                                     (global-variable? g.current name)
                                     (global-variable? g.init name) )
                                 #t #f ) )
                       (set! *pc* (stack-pop)) )
                     (signal-exception
                      #f (list "Not a variable name" name) ) )
                 (signal-exception
                  #t (list "Incorrect arity" 'variable-value) ) ) ) ) ) ) )
```

Функция `variable-defined?` служит для исследования реифицированных окружений. Она определяет, можно ли пользоваться какой-либо переменной в данном окружении. Этот вопрос не такой простой, каким кажется, так как если локальные окружения однозначно можно расширять, то для глобального окружения это не всегда так. Если глобальное окружение неизменяемо, то эта функция является чистой: ведь если переменной нет в глобальном окружении сейчас, то она не появится и потом. Однако, если глобальное окружение может измениться, то точно так же может измениться и ответ `variable-defined?`.

8.7. Рефлексивный интерпретатор

В середине восьмидесятых годов весьма модной была тема рефлексивных интерпретаторов — околобезумная фантазия исследователей, подарившая миру термин «рефлексивные башни». Представьте себе топь, укрытую туманом, а посреди неё тянущуюся ввысь башню, вершина которой растворяется в сером пасмурном небе, — прямо как у Рэкхема! Подобные мистические ассоциации проистекают из первых экспериментов с продолжениями, реификацией окружений и FEXPR диалекта Interlisp (над последними Кент Питман в [Pit80] провёл хорошую расстановку точек).

Ну кто из нас не мечтал создать (или хотя бы иметь) язык, где возможно переопределить всё, где нет предела воображению, где любые идеи могут свободно скакать по райским садам, не встречая помех и преград? К сожалению, сладкие грёзы уступают место суровой реальности, где мы получаем невыносимо медлительные системы, практически не поддающиеся компиляции и лишённые каких-либо твёрдых законов функционирования.

Как бы то ни было, в этом разделе рассматривается небольшой рефлексивный интерпретатор, мало что оставляющий недоступным или неявным. Естественно, он является далеко не первым в своём роде, например: [dRS84], [FW84], [Wan86], [DM88], [Baw88], [Que89], [IMY92], [JF92] и множество других. Все эти интерпретаторы отличаются друг от друга конкретными аспектами рефлексии и принципами реализации.

Рефлексивный интерпретатор обязан позволять интроспекцию, то есть давать программам возможность в любой момент времени захватывать контекст вычислений. Под контекстом вычислений здесь понимаются продолжение и лексическое окружение. Текущее продолжение можно получить с помощью знакомой вам функции `call/cc`. Для получения требуемого окружения давайте используем рассмотренную ранее форму `(export)`, также известную как функция `the-environment`, которая реифицирует текущее лексическое окружение. Реификация, осознание понятий является необходимым требованием рефлексии, но существует множество способов её проведения, и выбор способа влияет на доступные в будущем возможности.

Как сказано в [Chr95]: «Puisqu'une fois la borne franchie, il n'est plus de limite». И правда, ограничения бессмысленны для тех, кто уже вышел за пределы дозволенного, поэтому мы дадим пользователям возможность определять собственные специальные формы. Interlisp экспериментировал с механизмом, присутствовавшим в Lisp 1.5 под названием FEXPR. При вызове эта форма получала не значения аргументов, а буквально их текст вместе с текущим лексическим окружением. После этого она могла вычислять их в любом необходимом порядке, или, обобщая, делать с ними вообще всё, что ей заблагорассудится. Мы введём подобную ей специальную форму `flambda` со следующим синтаксисом:

`(flambda (`*переменные*`...) `*формы*`...)`

Первая переменная связывается с текущим лексическим окружением, а все остальные — с текстом аргументов, переданных при вызове. Например, вот так через `flambda` определяется цитирование:

```
(set! quote (flambda (r quotation) quotation))
```

Естественно, рефлексивный интерпретатор должен позволять изменение и самого себя (захватывающая возможность, несомненно), поэтому мы сделаем все его внутренние функции доступными интерпретируемым программам. Функция `the-environment` будет ответственной за помещение данных функций в глобальное окружение.

Итак, вот он — рефлексивный интерпретатор. Он занимает всего лишь 1362 байта[14] — достаточно мало, чтобы с лёгкостью поместиться в стандартную библиотеку. Интерпретатор написан языком, доступным нашему компилятору в байт-код.

```
(apply
 (lambda (make-toplevel make-flambda flambda? flambda-apply)
   (set! make-toplevel
     (lambda (prompt-in prompt-out)
       (call/cc
        (lambda (exit)
          (monitor (lambda (c b) (exit b))
            ((lambda (it extend error global-env toplevel
                      eval evlis eprogn reference )
               (set! extend
                     (lambda (env names values)
                       (if (pair? names)
                           (if (pair? values)
                               ((lambda (newenv)
                                  (begin
                                   (set-variable-value!
                                    (car names) newenv (car values) )
                                   (extend newenv (cdr names)
                                           (cdr values) ) ) )
                                (enrich env (car names)) )
                               (error "Too few arguments" names) )
                           (if (symbol? names)
                               ((lambda (newenv)
                                  (begin
                                   (set-variable-value!
                                    names newenv values )
                                   newenv ) )
                                (enrich env names) )
                               (if (null? names)
                                   (if (null? values) env
                                       (error "Too many arguments"
                                              values ) )
                                   env ) ) ) ) )
               (set! error
                     (lambda (msg hint)
                       (exit (list msg hint)) ) )
               (set! toplevel
                     (lambda (genv)
                       (set! global-env genv)
```

[14] В откомпилированном виде, конечно же, потому как его исходный код содержит чуть менее 3000 символов, плюс ещё столько же пробелов.

```
        (display prompt-in)
        ((lambda (result)
           (set! it result)
           (display prompt-out)
           (display result) (newline) )
         ((lambda (e)
            (if (eof-object? e) (exit e)
                (eval e global-env) ) )
          (read) ) )
        (toplevel global-env) ) )
(set! eval
      (lambda (e r)
        (if (pair? e)
            ((lambda (f)
               (if (flambda? f)
                   (flambda-apply f r (cdr e))
                   (apply f (evlis (cdr e) r)) ) )
             (eval (car e) r) )
            (if (symbol? e) (reference e r) e) ) ) )
(set! evlis
      (lambda (e* r)
        (if (pair? e*)
            ((lambda (v)
               (cons v (evlis (cdr e*) r)) )
             (eval (car e*) r) )
            '() ) ) )
(set! eprogn
      (lambda (e+ r)
        (if (pair? (cdr e+))
            (begin (eval (car e+) r)
                   (eprogn (cdr e+) r) )
            (eval (car e+) r) ) ) )
(set! reference
      (lambda (name r)
        (if (variable-defined? name r)
            (variable-value name r)
            (if (variable-defined? name global-env)
                (variable-value name global-env)
                (error "No such variable" name) ) ) ) )
((lambda (quote if set! lambda flambda monitor)
   (toplevel (the-environment)) )
 (make-flambda
  (lambda (r quotation) quotation) )
 (make-flambda
  (lambda (r condition then else)
    (eval (if (eval condition r) then else) r) ) )
```

```
                (make-flambda
                 (lambda (r name form)
                   ((lambda (v)
                      (if (variable-defined? name r)
                          (set-variable-value! name r v)
                          (if (variable-defined? name global-env)
                              (set-variable-value! name global-env v)
                              (error "No such variable" name) ) ) )
                    (eval form r) ) ) )
                (make-flambda
                 (lambda (r variables . body)
                   (lambda values
                     (eprogn body (extend r variables values)) ) ) )
                (make-flambda
                 (lambda (r variables . body)
                   (make-flambda
                    (lambda (rr . parameters)
                      (eprogn body
                              (extend r variables
                                      (cons rr parameters) ) ) ) ) ) )
                (make-flambda
                 (lambda (r handler . body)
                   (monitor (eval handler r)
                     (eprogn body r) ) ) ) ) )
           'it 'extend 'error 'global-env 'toplevel
           'eval 'evlis 'eprogn 'reference ) ) ) ) ) )
  (make-toplevel "?? " "== ") )
'make-toplevel
((lambda (flambda-tag)
   (list (lambda (behavior) (cons flambda-tag behavior))
         (lambda (o) (if (pair? o) (= (car o) flambda-tag) #f))
         (lambda (f r params) (apply (cdr f) r params)) ) )
 98127634 ) )
```

Как сказала бы Юлия Кристева,[15] данное определение пропитано изощрённостью как минимум в плане используемых обозначений. Вероятно, вам придётся сначала немного поработать с этим интерпретатором, прежде чем вы уверуете в него.

Внешняя форма `apply` создаёт четыре локальные переменные. Последние три из них отвечают за рефлексивные абстракции, `flambda`-формы, а именно: `make-flambda` создаёт их; `flambda?` опознаёт их; `flambda-apply` применяет их. Эти абстракции обладают весьма своеобразным протоколом вызова, так что у вас может уйти некоторое время на его понимание. Первая переменная, `make-toplevel`, инициализируется в теле применяемой функции, дабы

[15] Ну, она сказала нечто подобное на радио France Musique тем воскресным утром 19 сентября 1993 года, когда я писал эти строки.

она могла сделать доступными пользователям все четыре локальные переменные. По завершении инициализации она запускает интерактивную сессию с настраиваемыми приглашениями ко вводу. Изначально это `??` и `==`.

Перед началом инициализации захватывается продолжение — оно будет активировано в случае возникновения ошибки или при прямом вызове функции `exit`.[16] Данное продолжение также доступно интерпретируемым программам. Цикл, реализующий интерактивную сессию, защищён от необработанных исключений формой `monitor`. [**см. стр. 308**] Далее следуют объявление и инициализация служебных переменных и функций, которые, естественно, будут доступны и пользователям. Переменная `it` связывается со значением последней вычисленной формы. Функция `extend`, конечно же, расширяет окружение новыми привязками. Функция `error` печатает сообщение об ошибке и завершает работу программы, вызывая `exit`.

Функция `toplevel`, как обычно, отвечает за диалог с пользователем. Вспомогательные для `eval` функции (`evlis`, `eprogn` и `reference`) имеют стандартные определения, разве что теперь они доступны пользователям. Функция `eval` сейчас — сама простота. Вычисляемое выражение может быть или переменной, или неявно цитированным значением, или формой. В последнем случае вычисляется первый элемент списка и, если он оказался рефлексивной функцией, то ей передаётся текущее окружение вместе с параметрами вызова; если же это обычная функция, то она и вызывается как обычная функция.

Язык, определяемый данным интерпретатором, невозможно эффективно компилировать из-за его невероятной гибкости. Предположим, мы определили следующую нормальную функцию:

```
(set! stammer (lambda (f x) (f f x)))
```

Теперь посмотрим, как она себя ведёт с рефлексивными абстракциями:

```
(stammer (lambda (ff yy) yy) 33)  → 33
(stammer (flambda (ff yy) yy) 33) → x
```

Получается, что первый вызов `stammer` применяет переданную функцию саму к себе, а второй — реифицирует своё собственное тело и применяет функцию *к самому себе*. Следовательно, в присутствии `flambda` необходимо компилировать все аппликации каждой абстракции двумя различными способами: рефлексивным и обычным. И это касается не только функций, но и всех обращений к специальным формам — ведь теперь они тоже являются абстракциями, которые можно переопределить. Короче говоря, мы отобрали у компилятора практически все инварианты, на которые он может полагаться при оптимизации генерируемого кода, а в таких условиях интерпретация оказывается ничем не хуже.

Перед запуском главного цикла определяются несколько рефлексивных функций, просто потому, что их проще определить именно там. Таким образом

[16] Наконец-то у нас есть Лисп, из которого можно нормально выйти! Можете самостоятельно убедиться в том, что Common Lisp, Scheme и даже Dylan лишены такого удобства.

получаются `quote`, `if`, `set!`, `lambda`, `flambda` и `monitor`. Все остальные можно добавить потом:

```
(set! global-env
      (enrich global-env 'begin 'the-environment) )
(set! the-environment
      (flambda (r) r) )
(set! begin
      (flambda (r . forms)
        (eprogn forms r) ) )
```

Переменная `global-env` связана с реифицированным окружением, содержащим все предшествующие определения, включая саму себя, — и это её главное достоинство. Глобальное окружение не только доступно интерпретируемым программам, но и произведённые ими изменения имеют обратную силу. Интерпретатор и исполняемые им программы пользуются одной и той же `global-env`. Действия любой стороны имеют одинаковые последствия, приносят те же радости и печали. Предыдущий пример работает именно благодаря этой взаимосвязи. Давайте рассмотрим ещё один, определяющий новую специальную форму `when`.

```
(set! global-env (enrich global-env 'when))
(set! when
      (flambda (r condition . body)
        (if (eval condition r) (eprogn body r) #f) ) )
```

Сначала глобальное окружение расширяется новой переменной, которая, конечно же, пока не имеет никакого значения. Сразу же после этого она инициализируется рефлексивной функцией. Так как глобальное окружение общее, то новая переменная видна в том числе и интерпретатору, и функция `eval` отныне будет вести себя с ней так же, как и с любой другой специальной формой.

Вероятно, вас всё ещё мучает вопрос: «Ну и при чём здесь та укрытая туманом башня, о которой говорилось в начале?» Просто представьте, что каждый её этаж — это рефлексивный интерпретатор. Программы, исполняемые интерпретаторами, могут пристраивать к башне новые этажи, создавать новые уровни интерпретации с помощью функции `make-toplevel`. Также они могут передать явному вычислителю своё собственное определение, добившись таким образом поразительного замедления работы, что скорее всего переносит подобные вещи в разряд мысленных экспериментов. Самоинтерпретация немного отличается от рефлексии; она требует строгого следования двум правилам. Во-первых, нельзя напрямую использовать специальные формы для определения других специальных форм. Надеюсь, вам понятно, что перестраивать фундамент башни, сидя при этом в ней же, — не самая хорошая идея. Именно поэтому абстракция, связывающая переменные вроде `quote` и `set!` с их определениями, имеет такое простое тело: `(toplevel (the-environment))`. Во-вторых, `flambda`-замыкания должны быть понятны всем

уровням интерпретации (вернее, любым двум соседним уровням). Поэтому соответствующие функции определяются отдельно и используют для идентификации специальную уникальную метку `flambda-tag`. Хотя, конечно же, при таком определении их легко можно обмануть.

В итоге программы получают возможность осознания собственных действий (включая их продолжение и окружение, в котором они выполняются), далее они могут проанализировать своё поведение, запрограммировать себя на что-то другое и продолжить работу по новому плану. Интроспекция такого уровня предоставляет поистине захватывающие возможности. Анализ и модификация контекста вычислений рассмотрены, например, в [Que93a]. Именно благодаря подобным возможностям эти интерпретаторы и называются *рефлексивными*, а Лисп сыскал славу языка искусственного интеллекта.

Форма `define`

Забавно, но предыдущий рефлексивный интерпретатор позволяет весьма просто определить довольно сложную форму — `define`. Как было сказано ранее, `define` — это *очень* специальная форма: она ведёт себя одновременно как определение и как присваивание. С одной стороны, после выполнения `define` переменная имеет чётко определённое значение. С другой стороны, ещё до выполнения тела `define` в окружении магическим образом появляется новая переменная, известная как внутри самой `define`, так и повсюду после неё. Все эти аспекты семантики специальной формы `define` хорошо видны в приведённом определении. Для краткости мы не стали реализовывать её чисто синтаксические возможности, позволяющие писать выражения вроде `(define (foo x) (define (bar) ...) ...)`, а также остальные правила преобразования вложенных форм `define`. (Серьёзно, `define` — это сложно.)

```
(set! global-env (enrich global-env 'define))
(set! define
      (flambda (r name form)
        (if (variable-defined? name r)
            (set-variable-value! name r (eval form r))
            ((lambda (rr)
               (set! global-env rr)
               (set-variable-value! name rr (eval form rr)) )
             (enrich r name) ) ) ) )
```

Прежде всего `define` проверяет, существует ли уже переменная. Если это так, то её (пере)определение, в соответствии с R^5RS, сводится к присваиванию нового значения. В противном случае в глобальном окружении создаётся новая переменная, и её начальное значение вычисляется в новом, расширенном окружении, позволяющем рекурсивные определения.

8.8. Заключение

В этой главе были рассмотрены различные аспекты использования явного вычислителя, касающиеся как обычных функций, так и специальных форм. В зависимости от желаемых качеств встроенного языка, возможны различные варианты реализации вычислителя, в частности, использующие полноценные окружения или квазистатические привязки. Это прекрасная иллюстрация того, что проектирование языков программирования действительно является искусством, предоставляющим истинно неограниченные возможности решения возникающих в процессе задач.

Также данная глава показывает, насколько повезло Лиспу (вернее, насколько удачные решения были приняты его автором, Джоном Маккарти) с представлением программ и данных, сближающим понятия языка и метаязыка, благодаря чему открывается широчайшее поле для экспериментов.

8.9. Упражнения

Упражнение 8.1 Почему функция `variables-list?` ничего не предпринимает во избежание зацикливания при обработке списка переменных?

Упражнение 8.2 Специальная форма `eval/ce` компилирует на лету передаваемые ей выражения. Но она делает это каждый раз заново, что не особо эффективно, если требуется вычислить одно и то же выражение несколько раз. Придумайте, как исправить этот недостаток.

Упражнение 8.3 Улучшите определение `eval/at` через `eval/ce`, избавившись от нечаянно захватываемой переменной. Подсказка: `gens...`

Упражнение 8.4 Может ли пользователь определить `variable-defined?` самостоятельно? Почему нет/как именно?

Упражнение 8.5 Переведите определение рефлексивного интерпретатора на стандартный Scheme.

Рекомендуемая литература

Статья [dR87] отлично показывает, насколько всё же Лисп рефлексивен. В [Mul92] рассматривается алгебраическая семантика его рефлексивных возможностей. Также стоит взглянуть на работы [JF92] и [IMY92], затрагивающие рефлексию в общем.

Глава 9

Макросы: употребление и злоупотребление

Презираемые, неверно используемые, несправедливо осуждаемые, недостаточно оправданные (теоретически) — как бы их не клеймили, макросы остаются одним из столпов Лиспа и одной главных из причин его долголетия. Если функции являются абстракцией вычислений, а объекты — абстракцией данных, то макросы абстрагируют структуру программ. В этой главе мы познакомимся с макросами и вызываемыми ими проблемами. Ввиду того, что макросы являются одной из наименее изученных тем в Лиспе, существует огромное разнообразие реализаций макросистем. Среди них сложно выделить канонические решения, поэтому кода в этой главе будет весьма умеренное количество, а больше внимания уделяется изучению эволюции этих своеобразных сущностей — макросов.

Изобретение макросов приписывается [SG93] Тимоти Харту, именно он придумал их в 1963 году вскоре после выхода руководства по Lisp 1.5; с тех пор и поныне они являются неотъемлемой частью диалектов Лиспа. Макросы позволяют разработчикам проектировать и реализовывать языки, соответствующие решаемым проблемам. Подобно математике, где обычным делом является введение обозначений для новых понятий, макросы позволяют расширять язык новыми синтаксическими конструкциями. Не поймите превратно: дело не в расширении возможностей языка с помощью библиотек функций или чего-то подобного. Лисп с графической библиотекой всё ещё остаётся Лиспом и не более чем Лиспом. Здесь же речь идёт об увеличении выразительной силы языка с помощью введения новых синтаксических форм.

Расширение языка подразумевает введение новой нотации, которая позволяет, например, писать $\lim_{x \to x_0} f(x) = A$, когда имеется в виду $\forall \varepsilon > 0 \colon \exists \delta = {} = \delta(\varepsilon) \colon \forall x \colon 0 < |x - x_0| < \delta \Rightarrow |f(x) - A| < \varepsilon$. Естественно, можно было бы каждый раз в деталях расписывать понятие предела, если бы не лень. Но лень — двигатель прогресса. Подробные определения являются лишь обузой для понимания более широких концепций, поэтому со временем заменяются подобной краткой и удобной формой записи. Очень многие математические понятия настолько глубоко проработаны, что ими совершенно невозможно

пользоваться в полностью развёрнутом виде, не прибегая к каким-либо сокращениям. Также ради гибкости часто вводятся параметры: ведь когда мы пишем $\lim_{x\to x_0} f(x) = A$, то на самом деле имеем в виду параметризованное сокращение $L(f(x), x_0, A)$. Макросы — это отнюдь не изящный хак, доступный лишь избранным, а невероятно полезная возможность, позволяющая строить абстракции на уровне программ, манипулируя их представлением.

В большинстве императивных языков набор синтаксических конструкций фиксирован. Например, в них наверняка есть цикл `while`, кое-где есть и `until`, но если вам нужен какой-то особый цикл, то чаще всего вы лишены возможности записывать его подобно другим циклам. Лисп же вполне возможно научить понимать `(repeat :while` *p* `:unless` *q* `:do` *тело*`...)` как

```
(let loop ()
  (if p (begin (if (not q) (begin тело...))
               (loop) )) )
```

Этот пример намеренно «расточителен»: в нём используются дополнительные ключевые слова (начинающиеся с двоеточия), хоть в Scheme и принято минимизировать подобный синтаксический шум. [**см. упр. 9.1**] Кроме того, данный макрос вводит локальную переменную `loop`, которая может скрыть одну из свободных переменных *p*, *q* или *тела*. Истинным любителям циклов стоит взглянуть на местный Эверест прогресса — макрос `loop`, реализация которого занимает не одну сотню строк, а описанию его посвящена целая глава стандарта: [Ste90, глава 26].

К сожалению, как и множество других непростых и хитроумных концепций, макросы легко могут стать неуправляемыми. Цель данной главы — осознать их недостатки и обнаружить их преимущества. Для этого мы попытаемся логически воссоздать макросы, чтобы добраться до корня их проблем и причин их разнообразия.

9.1. Подготовка

Рассмотренные в предыдущих главах интерпретаторы разделяют обработку программ на две фазы: *подготовку* (в терминологии ISLisp) и *исполнение*. Таким образом, общая схема вычислений похожа на быстрый интерпретатор [**см. стр. 222**]: `(run (prepare` *выражение*`))`. Нечто подобное выполняет и компилятор [**см. стр. 269**], который сначала преобразует программы в деревья промежуточных инструкций и только по ним уже генерирует байт-код. Более ранние реализации тоже вполне вписываются в эту теорию: достаточно считать подготовку тождеством.

Сама подготовка также может быть многофазной. Например, все считываемые выражения изначально являются лишь строками символов, и во многих диалектах Лиспа существует понятие макросимволов, способных влиять на процесс синтаксического анализа этих строк и построения соответствующих S-выражений. Известным примером такого символа является кавычка: она

означает, что следующее считанное выражение должно быть обёрнуто в форму `quote`. Можно представить, что внутри синтаксического анализатора кавычка вызывает выполнение выражения `(list (quote quote) (read))`.

Если считывателем Лиспа можно управлять, то функция `read` может адаптироваться под любые нужды. Например, Bigloo [SW94] при компиляции исходных файлов, написанных на Caml Light, просто использует соответствующую функцию для их чтения; единственное требование к ней — она должна возвращать корректные S-выражения. Фактически, данная функция `read` является фронтендом компилятора Caml Light [LW93].

Мы не будем особо касаться макросимволов, так как они всё же являются встроенной возможностью алгоритма чтения S-выражений, а не самостоятельной конструкцией.

Фаза подготовки нередко является составной частью фазы компиляции. В любом случае, важно чётко отделять эту фазу от следующей за ней фазы исполнения. Помните: тесным должно быть взаимодействие компонентов, а не зависимости между ними. На этой почве возникают два различных варианта понимания подготовки: *многомировой* и *унитарный*.

9.1.1. Множественные миры

В этом случае результаты подготовки выражений сохраняются в файлах (обычно `*.o` или `*.fasl`). Исполнять такие файлы помогает отдельное приложение — загрузчик, который нередко по совместительству также занимается управлением связями между независимо подготовленными выражениями, то есть является компоновщиком. Большинство популярных языков программирования исповедуют именно такую модель раздельной компиляции. Её преимуществом является возможность обработки программ по частям, а также более естественное управление пространствами имён с помощью разнообразных директив экспорта и импорта. Мы говорим о множественных мирах, так как подготовленные выражения — это единственное средство связи между предобработчиком программ и их исполнителем. Исходные коды программ и исполняемые программы живут в отдельных мирах, которые лишены возможности влиять друг на друга.

$$\left.\begin{array}{lcl} \textit{выражение}_0 & \xrightarrow{\texttt{prepare}} & \textit{подготовленное-выражение}_0 \\ \textit{выражение}_1 & \xrightarrow{\texttt{prepare}} & \textit{подготовленное-выражение}_1 \\ \ldots & \xrightarrow{\texttt{prepare}} & \ldots \\ \textit{выражение}_n & \xrightarrow{\texttt{prepare}} & \textit{подготовленное-выражение}_n \end{array}\right\} \texttt{run}$$

9.1.2. Единый мир

В противоположность многомировой гипотезе, адепты секты единого мира считают центром вселенной интерактивную сессию, поэтому все вычисления,

начавшиеся в ней, совместно существуют в общей памяти, где нет никаких преград для общения посредством глобальных разделяемых ресурсов (переменных, списков свойств и т. п.). Интерактивная сессия связывает воедино процессы чтения выражений, их подготовки и исполнения. Она часто предстаёт в виде командного интерпретатора, контролирующего данные процессы. Заметьте, уникальность мира вовсе не означает строгую последовательность всех происходящих процессов: вполне возможно отложить исполнение обработанной программы на потом. Во многих лисп-системах существует функция `compile-file`, которая считывает выражения из файла и сохраняет результат компиляции в другой файл. Выполнить же подготовленные к исполнению программы можно потом, загрузив их с помощью функции `load` или чего-то подобного. Таким образом, и здесь имеется возможность подготавливать программы по частям.

Идея единого мира отнюдь не оторвана от реальности. Напротив, она является взглядом на программы как на среду исполнения кода, подобную операционным системам вроде UN⋆X, где пользователь рассчитывает на некоторое внутреннее состояние: файловую систему, всевозможные переменные окружения, псевдонимы команд и т. д., а управляет всем этим с помощью командного интерпретатора вроде `bash` или `zsh`.

Как бы то ни было, одним из основных требований к программным системам является предсказуемость и воспроизводимость результатов. Ожидается, что из одного и того же исходного кода при одинаковых условиях получается одинаковый исполнимый код. Это базовая предпосылка, без которой невозможно нормальное использование данной системы. Множественные независимые миры без проблем ей удовлетворяют: достаточно проконтролировать все входные данные, а кроме них ничто больше не может повлиять на систему. В случае единого мира неучтённым остаётся его внутреннее состояние[1] вместе со всевозможными способами его изменения. Такой компилятор постоянно остаётся в памяти, понемногу накапливает вносимые в него изменения, и мы постепенно теряем над ним контроль.

Подводя итог, подготовка программ обязана быть полностью контролируемым процессом.

9.2. Раскрытие макросов

Итак, обработка сокращений-макросов выполняется во время подготовки программ, перед их исполнением. Следовательно, логично будет разделить эту подготовку на две части: 1) *раскрытие* макросов, 2) всё остальное. То есть выполнять её как-то так: `(really-prepare (macroexpand` *выражение*`))`. Нас интересует вот эта функция-*экспандер* `macroexpand`, отвечающая за обработку

[1] Например, никто не может сказать загодя, какие переменные окружения выведет `printenv`, или что написано в конфигурационных файлах операционной системы.

макросов. Как именно задаётся алгоритм, который она реализует? Возможны два способа её определения: *экзогенный* и *эндогенный*.

9.2.1. Экзогенный подход

Экзогенное раскрытие, как его описывают [QP91b, DPS94b], подразумевает, что функция `macroexpand` предоставляется *отдельно* от раскрываемого выражения, например, с помощью специальных директив.[2] Следовательно, функция `prepare` в общих чертах будет выглядеть так:

```
(define (prepare expression directives)
  (let ((macroexpand (generate-macroexpand directives)))
    (really-prepare (macroexpand expression)) ) )
```

Возможны некоторые вариации этой идеи, которые мы разберём на примере компилятора в байт-код. [**см. стр. 316**] Предположим, в нашем распоряжении есть загрузчик по имени `run` и компоновщик `build-application`. Кроме того, пусть имеется исполнимый файл `compile.so`, содержащий компилятор, а экспандеры будем называть `expand.so`.

Раскрытие макросов может происходить каскадом, как показано на верхней части рисунка 9.1. В таком случае вызов «`compile file.scm expand.so`» соответствует следующей командной строке Un⋆x-shell:

```
run expand.so < file.scm | run compile.so > file.so
```

Также макросы может раскрывать новый, только что синтезированный компилятор (здесь это `tmp.so`), как на нижней половине рисунка 9.1. Тогда команде «`compile file.scm expand.so`» соответствует

```
build-application compile.so expand.so -o tmp.so;
run tmp.so < file.scm > file.so
```

Среди нерешённых проблем остаётся ещё протокол обмена информацией между экспандером и компилятором. Экспандер принимает исходную программу и возвращает её с раскрытыми макросами. Он может принимать её буквально как значение, или же как имя файла, в котором она хранится. Аналогично, возвращать результат экспандер может как в виде непосредственных S-выражений, так и записав его в файл и вернув имя этого файла. Обмениваться информацией посредством файлов ничуть не абсурдно. Компилятор того же языка Си в процессе работы хранит готовые объектные модули в файлах. Подобный подход позволяет избежать скрытой передачи информации, так как любое общение выполняется явно и прозрачно через файлы. В случае обмена информацией посредством S-выражений, для подготовки потребуется динамически загружать экспандер в память с помощью `load` или чего-то подобного, потому как обмен информацией между программами выполняется напрямую.

[2] Располагаемых, к примеру, в самом начале файла.

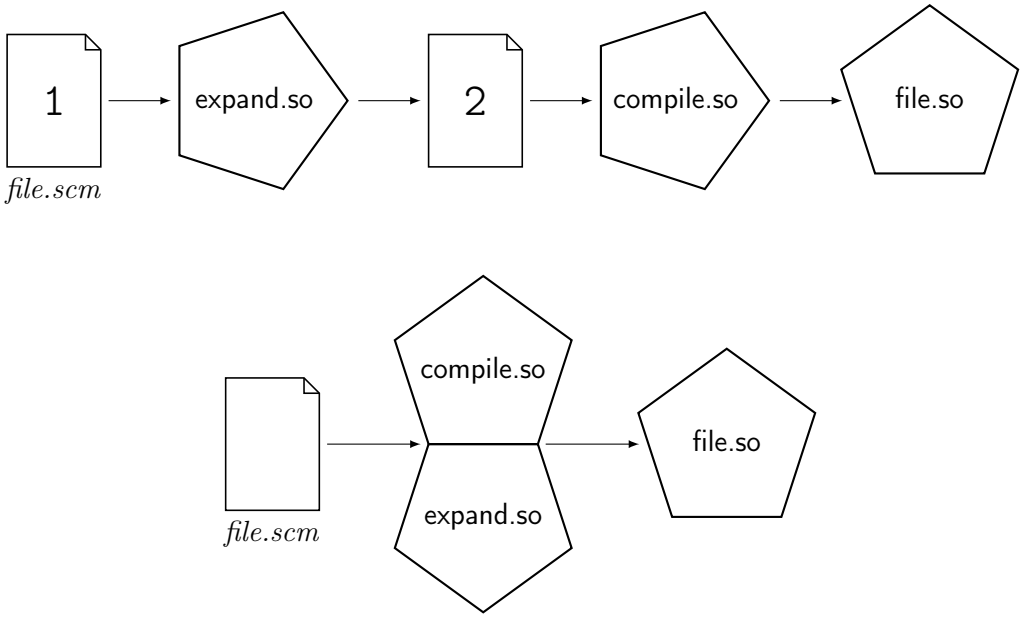


Рис. 9.1. Два варианта экзогенной компиляции. Пятиугольники обозначают исполнимые бинарные файлы.

Также экспандер не обязательно должен быть монолитным, его вполне можно собрать как конструктор из меньших блоков, обрабатывающих специфичные макросы. Для этого потребуется расширить язык директив, чтобы он мог оперировать подобными блоками. Ещё это означает, что макросы должны быть свободно компонуемыми, что в свою очередь приводит к необходимости обеспечения ассоциативности и коммутативности макроопределений.

Итого, экзогенный подход подразумевает использование отдельного независимого макроэкспандера для обработки программ. В этом случае мы несомненно обладаем полным контролем над процессом раскрытия макросов, а также можем быть уверены в том, что в готовых программах не останется никакого мусора после их раскрытия. Кстати, полное отделение раскрытия макросов от исполнения программ снимает любые ограничения на форму и реализацию экспандеров, так как абсолютно не важно, как они работают внутри, лишь бы возвращали корректные S-выражения. Вполне можно писать экспандеры на Си++, M4, да хоть на Perl! Тем не менее, Лисп кажется наиболее адекватным языком для описания алгоритмов обработки S-выражений. Как видим, экзогенный подход не подразумевает сколь-либо тесной связи между языком обрабатываемых программ и языком описания макросов. Макроэкспандеру необходимо только уметь самостоятельно определять в обрабатываемой программе местонахождение макросов, требующих раскрытия.

9.2.2. Эндогенный подход

Эндогенное раскрытие подразумевает, что вся информация, необходимая для обработки макросов, содержится *в самой* программе. Другими словами, подготовка программ выглядит так:

```
(define (prepare expression)
  (really-prepare (macroexpand expression)) )
```

Фундаментальным отличием эндогенного подхода от экзогенного является предопределённость алгоритма раскрытия макросов; им можно лишь управлять в той или иной мере. Также задача осложняется тем, что при обработке программы требуется отыскать в тексте не только макросы для их раскрытия, но и сами правила раскрытия. Эти правила обычно задаются специальными S-выражениями, чаще всего формами `define-macros` или `define-syntax`. Подобные определения ещё предстоит превратить в функции-экспандеры, обрабатывающие макросы соответствующим образом. Так, стоп, это что-то знакомое: мы динамически превращаем исходный текст в исполнимую программу… точно, `eval`! В таком подходе определённо есть нечто гениальное: не изобретать специальный новый язык, а использовать для описания макросов уже готовый Лисп!

Конечно, как и любая гениальная идея, она имеет свои подводные камни. В тёмные времена одной из сложнейших для освоения дисциплин оккультных компьютерных искусств была макрология.* Фактически, именно это отождествление языков затрудняет понимание истинной природы макросов. Долгое время они считались чем-то вроде особых функций, имеющих странный протокол вызова, в соответствии с которым вычислению подлежали не аргументы, а возвращаемое значение. Эта модель, несмотря на всё её удобство при интерпретации, была отвергнута ввиду сложностей, которые она вызывает для компиляторов, а также по иным причинам, описанным Кентом Питманом в статье [Pit80]. В современном понимании, макросы — это правила преобразования определённых сокращений в соответствующие им полные конструкции целевого языка.

Выбор Лиспа в качестве языка описания макросов оспаривается апологетами ограниченного подхода, считающими, что макросы должны уметь лишь манипулировать заранее написанным кодом, а не создавать его сами. Именно такие макросы реализуют `define-syntax`/`syntax-rules` из R^5RS. Конечно, в таком случае они обладают несколько меньшей выразительной силой, а некоторые преобразования не могут описать в принципе, но зато простые макросы (коих большинство) становятся гораздо понятнее. Несмотря на то,

* «Макрология, *сущ., ж.* (букв. *многословие* или *макрос* + *учение*) **1.** Набор обыкновенно сложных или хитроумных макросов, напр., как составная часть большой программной системы, написанной на Лиспе, TECO или (реже) ассемблере. **2.** Искусство и наука постижения смысла макрологий в значении 1 (ср. археология, экология, теология)» — The Jargon File.

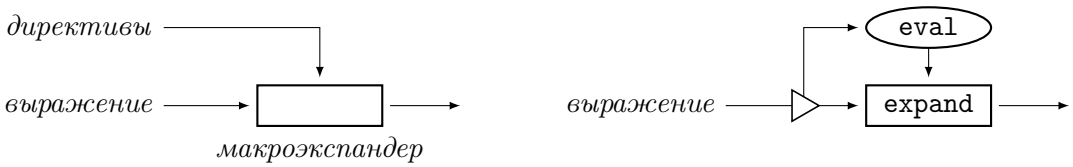


Рис. 9.2. Экзогенное (слева) и эндогенное (справа) раскрытие макросов.

что в данной книге более мощная форма `define-abbreviation`[3] используется чаще, лишь 5 макроопределений из порядка 50 используют её возможности, выходящие за рамки `define-syntax`. И даже из этих пяти только три макроса — фундаментальные определения MEROONET: `define-class`, `define-generic` и `define-method`), — принципиально нельзя реализовать по-другому.

Короче говоря, не важно, задаём ли мы алгоритм раскрытия явно или параметризуем готовый, в обоих случаях Лисп кажется наиболее подходящим языком для описания этого алгоритма. Но в отличие от экзогенного, эндогенный экспандер должен содержать внутри себя вычислитель; это его характерная особенность. Однако помните, что не следует смешивать язык описания макросов с языком обрабатываемой программы: они связаны, но не тождественны. В старых руководствах часто писалось, что макросы реализуются с помощью двукратных вычислений; их авторы грешат упущением того, что эти вычисления проводятся различными вычислителями.

9.3. Макровызовы

Макроэкспандер должен отыскать в обрабатываемой программе все макросы, требующие раскрытия. Некоторые языки (например, [Car93]) имеют тонко настраиваемые синтаксические анализаторы, позволяющие дополнять грамматику языка новыми формами напрямую. Добавить макрос там так же легко, как описать новый оператор: всего пара строк с его символом и арностью, информацией о том, префиксный он, инфиксный или постфиксный, плюс указание его приоритета.

Программы на Лиспе состоят из форм, записываемых S-выражениями, где основную смысловую нагрузку несёт их `car` — функциональный терм, — поэтому часто применяется следующий подход: список, первый элемент которого является ключевым словом, считается вызовом одноимённого макроса. У такого подхода есть множество преимуществ: он прост, расширяем и согласуется со всем остальным синтаксисом. Имя сокращения (макроса) ассоциируется с функцией (экспандером). В итоге получается весьма простой алгоритм раскрытия макросов: рекурсивно обходим обрабатываемое выражение в поисках ключевых слов; если натыкаемся на такое слово, то определяем со-

[3] Столь экзотическое имя выбрано с целью избежать противоречивых ассоциаций с конструкцией `define-macros`, присутствующей во многих реализациях Лиспа и Scheme.

ответствующий ему экспандер и применяем его к S-выражению, требующему раскрытия. В `cdr` этого S-выражения находится всё, что надо экспандеру для успешной работы.

Также можно вспомнить о макросимволах, присутствующих, например, в Common Lisp. Они связывают вызов экспандера с одним символом, а не с целой формой. Это позволяет избавиться от излишних скобок, если макрос применяется к чему-то короткому вроде имени переменной.

Таким образом, макрос можно понимать как функцию-экспандер, связанную с некоторым именем, приводящим к её вызову. Похоже, для подобных привязок потребуется собственное пространство имён. Однако помните, что макрос — это не привязка, функция или ключевое слово сами по себе, это триединая сущность внутри макроэкспандера. Кстати, раз это всё же пространство имён, нелишним будет иметь возможность определять как глобальные макросы, так и локальные: например, с помощью `define-abbreviation` и `let-abbreviation`.

Задачей функции-экспандера является превращение S-выражения в корректную программу. Следовательно, она ищет макросы в S-выражениях, а не программах. Конечно, некоторые части S-выражений действительно являются программами; да и синтаксис вызова макросов напоминает вызов функции. Но по сути это лишь S-выражения. Ничто не запрещает экспандеру трактовать некорректную форму `(foo . 5)` как сокращение для `(vector-ref foo 5)`!

Проблема здесь в том, что грамматика S-выражений гораздо менее ограничена, чем грамматика программ. Это вызывает некоторые затруднения, касающиеся приоритета макросов над другими конструкциями. Если `bar` это макрос, а `lambda` нет, то требует ли раскрытия форма `(bar 34)` в выражении `((lambda (bar) (bar 34)) ...)`? Здесь источником проблем является локальная переменная `bar`. Должна ли она скрывать одноимённый макрос `bar`? R^{5}RS, к примеру, считает, что должна. Тот же вопрос касается цитат: надо ли раскрывать `bar` в выражении `(quote (bar 34))`?

Таким образом, в алгоритм обхода S-выражений наверняка потребуется внести определённые доработки, дабы ограничить места, где допустимо применение макросов. Например, можно ли писать `(let (foo) ...)`, где `(foo)` — это макровызов, генерирующий список локальных переменных? Или, к примеру, `(cond (foo) ...)`, или `(case key (foo) ...)`? В Лиспе правила хорошего тона предписывают определять и использовать макросы таким образом, чтобы они были максимально похожи на функции. Тогда не особо важно, что именно скрывается за `(bar ...)` — макрос или функция.

9.4. Экспандеры

А что насчёт экспандеров? Какой контракт должны соблюдать они? Есть по меньшей мере два возможных варианта.

Классический подход подразумевает, что выражения, возвращаемые экспандером, всё ещё могут содержать нераскрытые макросы. Поэтому их необходимо повторно прогонять через другие экспандеры до полного раскрытия. Если функция `find-expander` возвращает экспандер, соответствующий имени макроса, то макровызов `(foo . `*expr*`)` раскрывается следующим образом:

`(macroexpand '(foo . `*expr*`)) ⇝`
`  ⇝ (macroexpand ((find-expander 'foo) '(foo . `*expr*`)))`

Есть и другой, более сложный подход, называемый в [DFH88] «*стилем передачи раскрытий*» (expansion passing style, EPS) по аналогии со стилем передачи продолжений. Идея в том, что экспандер обязан вернуть полностью готовое выражение, лишённое нераскрытых макросов. Таким образом, рекурсивные вызовы `macroexpand` снаружи переносятся внутрь самих экспандеров. Для этого, очевидно, экспандеры должны сами вызывать функцию `macroexpand`, а значит, им необходимо как-то получить к ней доступ. Можно было бы просто сделать `macroexpand` глобальной, но есть более искусное решение, которое и является сутью EPS. Можно передавать `macroexpand` как аргумент экспандерам, что одновременно решает проблему рекурсивных вызовов, а также значительно облегчает введение локальных макроопределений. В терминах Лиспа это выглядит так:

`(macroexpand '(foo . `*expr*`)) ⇝`
`  ⇝ ((find-expander 'foo) '(foo . `*expr*`) macroexpand)`

Эти два подхода к раскрытию макросов — классический и EPS — не эквивалентны. Очевидно, EPS более мощный, так как классическое раскрытие можно выразить через EPS, но не наоборот. Существенным отличием между ними является способ реализации локальных макросов. Допустим, мы хотим определить макрос, локальный для какого-либо подвыражения. В EPS достаточно при обработке этого подвыражения передать экспандеру расширенный вариант `macroexpand`, учитывающий новые определения. В языках, реализующих классический подход (в Common Lisp, к примеру) эта задача решается сложнее: специальный макрос `macrolet` изменяет внутреннюю структуру `macroexpand`, добавляя туда локальные определения, затем он выполняет раскрытие подвыражения, после чего вновь изменяет структуру `macroexpand`, убирая оттуда введённые ранее определения. Форма `macrolet` является примитивом в том смысле, что её невозможно определить самостоятельно, если её нет в языке изначально. Более того, вся система становится гораздо более хрупкой, если не только `macrolet` позволено модифицировать `macroexpand`. EPS не ведомы эти проблемы, так как вводимые локальные макросы всегда остаются локальными и могут свободно скрывать и переопределять внешние. Всё зависит только от новой функции, которая будет передана внутрь вместо `macroexpand`. В [DFH88] приведено множество примеров применения данного подхода для реализации каррирующих и трассирующих макросов.

Большая часть макросов действует локально, то есть выполняет простую замену одного текста другим. Важным свойством EPS является возможность

лёгкой модификации текущего экспандера, что позволяет выражать некоторые трансформации проще, чем при классическом подходе. Например, вспомните преобразование программ в «коробочный стиль». [**см. стр. 145**] Это преобразование легко реализуется с помощью двойного обхода кода: на первом проходе составляем список локальных изменяемых переменных, а на втором заменяем все обращения к ним соответствующими операции с коробками. EPS позволяет записать это легко и просто, буквально используя два разных экспандера: первый ничего не раскрывает, а только запоминает информацию для второго, который потом выполнит всё работу. Классический же подход и его намертво вшитый экспандер не позволяют такой гибкости, так что обход кода приходится реализовывать вручную, а не использовать уже готовую инфраструктуру.

Однако, EPS не всемогущ; существуют трансформации, которые и ему не по силам. [**см. стр. 176**] Перенос цитат не является локальным действием, так как здесь требуется вместо цитат подставить обращения к глобальным переменным. С заменой выражений проблем-то нет, но ведь эти переменные ещё надо создать (вставив глобальные `define` в подходящем месте), а это мало похоже на локальные изменения! Правда, здесь можно выкрутиться, обернув всю программу в макрос (скажем, `with-quotations-extracted`), который раскрывается в определения значений цитат, за которыми следует обработанная программа.

Существуют и другие макросы, которым необходимо создавать глобальные переменные, например, макрос `define-class` в MEROONET. Синтаксис Scheme не запрещает его использование внутри `let`-форм, но семантика данного макроса подразумевает создание глобальной переменной, которая содержит объект, описывающий определяемый класс. Такие задачи обычно нельзя решить, оставаясь в рамках макросистемы, так что `define-class` вынужден быть или специальной формой, или же макросом, использующим внутренние (intrinsic) функции реализации для достижения необходимого эффекта.

Однако, если EPS настолько хорош, то почему он так редко используется? Во-первых, его сложнее реализовать, но главная причина не в этом. Вспомните, что большая часть макросов проста по своей сути, а значит, более мощный и сложный механизм их раскрытия по сути ничего не даст, кроме замедления работы. В действительности, классический подход обладает существенным преимуществом: он однопроходен. Однажды раскрытые выражения, не начинающиеся на ключевое слово, больше никогда не будут просматриваться, так как гарантированно являются вызовами функций. EPS же позволяет экспандерам просматривать обрабатываемое выражение несколько раз, руководствуясь различными правилами раскрытия, что приводит к замедлению работы и усложнению понимания выполняемых ими трансформаций.

9.5. Приемлемость результатов раскрытия

Одним из пунктов контракта экспандеров является требование, чтобы возвращаемый ими результат был корректной программой, готовой для последующей предобработки. В программе не должно быть нераскрытых макросов; иными словами, это должна быть такая же программа, как если бы она была набрана вручную. На пути к этой светлой цели нас подстерегают несколько ловушек. Во-первых, раскрытие макросов — это некоторые вычисления, а вычисления иногда могут никогда не заканчиваться, что, согласитесь, несколько мешает получению готовых программ. Загнать компилятор в бесконечный цикл непросто, но если вам удалось, то понимайте это как плату за истинно неограниченные выразительные возможности макросистемы.

Один из лёгких способов попасть в бесконечный цикл — попытаться наивно определить макрос через самого себя. Как, например, в следующем определении макроса `while`:

```
(define-abbreviation (while condition . body)
  `(if ,condition (begin (begin . ,body)
                         (while ,condition . ,body) )) )
```

Помните, что в конце концов экспандер должен вернуть нормальную программу, поэтому после первого раскрытия макроса `while` он продолжит раскрывать макросы в получившемся коде, а там его ждёт ещё один `while`, а внутри него — ещё один, и так далее.

В Common Lisp допустить аналогичную ошибку гораздо проще благодаря ключевому слову `&whole`. С его помощью макрос может получить исходную форму, которая вызвала его раскрытие, но её недальновидное использование приводит к печальным последствиям.

Common Lisp
```
(defmacro while (&whole call)
  (let ((condition (cadr call))
        (body      (cddr call)))
    `(if ,condition (begin (begin . ,body) ,call)) ) )
```

Многие интерпретаторы выполняют раскрытие макросов на лету, сразу же после считывания S-выражений. При этом часто с целью оптимизации все макровызовы физически заменяются результатами раскрытия: это позволяет не дёргать макроэкспандер при каждом вызове функции, содержащей макросы. В таком случае предыдущий макрос `while` сгенерировал бы циклическое S-выражение, которое, в принципе, будет понятно интерпретаторам, но может вызвать затруднения у компиляторов, так как обычно они рассчитаны на работу исключительно с направленными ациклическими графами (деревьями, ветви которых могут сращиваться) [Que92a]. После преобразования макрос `while` будет выглядеть так:

Common Lisp
```
(defmacro while (&whole call)
  (let ((condition (cadr call))
        (body      (cddr call)) )
```

```
(setf (car call) 'if)
(setf (cdr call) `(,condition (begin (begin . ,body) ,call)))
call ) )
```

Подобные проблемы с циклами возможны и в цитатах; они тоже могут вынуждать некоторые компиляторы надолго задуматься. [**см. стр. 175**]

Ещё одним источником ошибок, часто даже более опасным, чем циклы, могут быть макросы, содержащие значения, вычисляемые во время раскрытия. Порядочные программисты строго соблюдают обет: никогда не писать макросы, раскрывающиеся в код, который невозможно написать вручную. Такое соглашение подразумевает наличие у программ некоторой письменной формы. В Scheme данная форма называется *внешним представлением*; любые цитаты (и вообще всё, что может прочитать функция `read`) обязаны записываться во внешнем представлении.

Давайте взглянем на особо поучительный пример цитирования значения, не имеющего внешнего представления. Следующий макрос возвращает процитированное продолжение собственного раскрытия:

```
(define-abbreviation (incredible x)
  (call/cc (lambda (k) `(quote (,k ,x)))) )
```

И что, по-вашему, эта тарабарщина означает? Давайте подумаем логически. Записать результат раскрытия в файл нельзя, так как у продолжений нет внешнего представления. Активировать это продолжение тоже вряд ли получится… Что вообще должно происходить при активации продолжения, захваченного *в другом* процессе? Ведь это то же самое, что во время работы приложения `a.out` начать выполнять нечто, что делалось после вызова компилятора «`cc main.c`», результатом работы которого стало данное приложение. Конечно, в некотором смысле `call/cc` можно считать машиной времени, но не буквально же! Раз на эти вопросы нельзя дать однозначные ответы, то нам остаётся лишь воспользоваться своим правом разработчика и объявить подобные программы ошибочными.

Не только продолжения приводят к таким курьёзам. Напрямую нельзя также записывать примитивы, замыкания, порты ввода-вывода. Всё это следует запретить; даже если `('(lambda (y) car) x)`, или `(`(,car x)`, или `(f ',(current-output-port))` не вызывают никаких проблем у интерпретаторов.

И напоследок, золотое правило макросов ещё раз: не генерировать программы, которые нельзя написать вручную.

9.6. Определение макросов

Существует множество форм, определяющих глобальные и локальные макросы (об их области видимости мы поговорим позже, в разделе 9.7). В Scheme это `define-syntax`, `letrec-syntax` и `let-syntax`; в Common Lisp — `defmacro` и `macrolet`; в этой книге — `define-abbreviation` и `let-abbreviation`.

Определить макрос — значит создать соответствующий ему экспандер и зарегистрировать его в качестве обработчика данного макроса. Код экспандеров записывается на специальном макроязыке, обычно диалекте Лиспа. Теперь давайте проанализируем, как различные миры и подходы, рассмотренные ранее, влияют на процесс определения макросов.

9.6.1. Множественные миры

Напомним, здесь раскрытие макросов и исполнение результатов раскрытия происходят в отдельных, непересекающихся мирах, не имеющих общей памяти.

Эндогенный подход

Эндогенный подход подразумевает, что определение макроса (в нашем случае это форма `define-abbreviation`) на лету переводится в исполнимый код экспандера с помощью явного вычислителя, реализующего макроязык. Таким образом, ключевое слово `define-abbreviation` — это синтаксический маркер для обработчика, указывающий местоположение макроопределений. Следующая программа является наивной иллюстрацией эндогенной стратегии раскрытия макросов:

```
(define *macros* '())

(define (install-macro! name expander)
  (set! *macros* (cons (cons name expander) *macros*)) )

(define (naive-endogenous-macroexpander exps)
  (define (macro-definition? exp)
    (and (pair? exp)
         (eq? (car exp) 'define-abbreviation) ) )
  (if (pair? exps)
      (if (macro-definition? (car exps))
          (let* ((def       (car exps))
                 (name      (car (cadr def)))
                 (variables (cdr (cadr def)))
                 (body      (cddr def)) )
            (install-macro! name (macro-eval
                                  `(lambda ,variables . ,body) ))
            (naive-endogenous-macroexpander (cdr exps)) )
          (let ((exp (expand-expression (car exps) *macros*)))
            (cons exp (naive-endogenous-macroexpander (cdr exps))) ) )
      '() ) )
```

Эта функция принимает список выражений (считанный из подготавливаемого файла) и последовательно их обрабатывает, глядя в глобальную переменную `*macros*`, которая хранит текущие объявленные макросы. Собственно

раскрытие выполняется функцией `expand-expression`. Если экспандер в процессе работы натыкается на определение макроса, то оно тут же обрабатывается и помещается в `*macros*`. За создание тела нового экспандера отвечает вычислитель `macro-eval`. Он может отличаться от обычного `eval`, ведь язык описания макросов не обязательно совпадает с целевым языком экспандера. Макросы и обрабатываемые ими программы принадлежат различным, абсолютно независимым мирам. Следующий пример отлично иллюстрирует это утверждение.

si/chap9b.scm

```
(define (fact1 n)
  (if (= n 0) 1
      (* n (fact1 (- n 1))) ) )

(define-abbreviation (factorial n)
  (define (fact2 n)
    (if (= n 0) 1
        (* n (fact2 (- n 1))) ) )
  (if (and (integer? n) (> n 0))
      (fact2 n)
      `(fact1 ,n) ) )

(define (some-facts)
  (list (factorial 5) (factorial (+ 3 2))) )
```

При эндогенном задании экспандеров нехорошо путать определения `fact1` и `fact2`, так как `fact1` для экспандера это всего лишь набор S-выражений, а не функция; такой функции во время раскрытия макросов вообще не существует. Поэтому `factorial` определяет необходимый ему `fact2` самостоятельно. В итоге, после раскрытия мы получаем следующее:

si/chap9b.escm

```
(define (fact1 n)
  (if (= n 0) 1
      (* n (fact1 (- n 1))) ) )

(define (some-facts)
  (list 120 (fact1 (+ 3 2))) )
```

Можно было бы сделать этот пример более запутанным, назвав `fact1` и `fact2` просто `fact`. Это немного затруднит восприятие кода `factorial` человеком, но не вызовет принципиальных проблем у экспандера, так как различные упоминания переменной `fact` чётко разделены: в одном месте это имя будет означать локальную переменную макроса, существующую только во время его раскрытия, а в другом — глобальную переменную `fact`, существующую во время исполнения программы.

Эндогенный подход, конечно, допускает некоторые вариации. Иногда раскрытие выполняется в два прохода: сначала из программы извлекаются все макроопределения, а на втором проходе уже выполняется раскрытие. Такой вариант может работать неправильно, если макросам позволяется определять другие макросы, потому что новые определения, получающиеся при раскрытии, могут быть проигнорированы. Также здесь возникает возможность использовать макросы до их определения в программе, что слегка сбивает с толку. При последовательной обработке слева направо подобные затруднения отсутствуют в принципе.

Другой интересной (и весьма часто встречающейся) вариацией является реализация `define-abbreviation` как примитивного макроса.

Экзогенный подход

При экзогенном подходе определения макросов отделены от обрабатываемой программы. Допустим, они могут быть отделены и друг от друга: то есть макросы можно независимо друг от друга определять в разных модулях, а не обязательно все в одном месте. Один из вариантов реализации такого поведения — специальный макрос, который определяет макросы. Назовём его `define-abbreviation`, вот его метациклическое определение:

```
(define-abbreviation (define-abbreviation call . body)
  `(install-macro! ',(car call) (lambda ,(cdr call) . ,body)) )
```

Задачей этого макроса является установка нового определения в макросистему. Пусть для этой цели определена функция `install-macro!`. Такой подход несколько усложняет раскрытие макросов, так как определяются они в одном месте, а используются в другом. Предположим, у нас есть следующий модуль:

si/chap9c.scm

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )

(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0))
      (fact n)
      `(fact ,n) ) )
```

После раскрытия макросов он выглядит так:

si/chap9c.escm

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

```
(install-macro! 'factorial
  (lambda (n) (if (and (integer? n) (> n 0))
                  (fact n)
                  `(fact ,n) )) )
```

Теперь используем макрос `factorial`, определяемый этим модулем, для раскрытия макросов в другом модуле, где определяется функция `some-facts`:

si/chap9d.scm

```
(define (some-facts)
  (list (factorial 5) (factorial (+ 2 3))) )
```

Мы получим следующий результат:

si/chap9d.escm

```
(define (some-facts)
  (list 120 (fact (+ 2 3))) )
```

Первая ссылка на `fact` в макросе `factorial` затруднений не вызывает, это очевидный вызов глобальной функции, определяемой в этом же модуле `si/chap9c.scm`. Второе упоминание имени `fact` происходит уже в модуле `si/chap9d.escm`. Вот и первая проблема: переменная `fact`-то свободна в макросе `factorial`: он на неё ссылается, но сам её ни с чем не связывает, а ведь такой переменной может и вовсе не быть в модуле `si/chap9d.escm`!

Простое решение заключается в подключении к `si/chap9d.scm` модуля, содержащего нужное определение. У нас пока есть только один подходящий модуль — `si/chap9c.scm`. Но если подключить его, то это приведёт к повторному определению макроса `factorial`, которому здесь не место. Более того, это определение будет ещё и ошибочным, если функция `install-macro!` в пользовательском коде означает что-то другое или вовсе не существует.

Попробуем улучшить данное решение. Для этого необходимо разобраться с зависимостями, возникающими между макросами и различными библиотеками; особенно, *особенно* с тем, на каких этапах исполнения программы они возникают. Макросы сложны отчасти потому, что требуют хорошего понимания последовательности всех выполняемых действий. Вернёмся к примеру с `fact1`, `fact2` и `factorial`. Во время раскрытия макроса `factorial` экспандеру необходима функция `fact2`. Результату же раскрытия для успешной работы необходима функция `fact1`. Будем говорить, что функция `fact1` принадлежит библиотеке *времени исполнения* макроса `factorial`, а `fact2` — библиотеке *времени раскрытия* `factorial`. Эти библиотеки полностью независимы. Действительно, библиотека времени раскрытия нужна только во время раскрытия, а библиотека времени исполнения — только во время исполнения кода уже раскрытого макроса.

Итак, решение сводится к тому, что макросы и их библиотека времени раскрытия определяются в одном модуле: `si/libexp`, а все функции библио-

теки времени исполнения — в другом: `si/librun`. [**см. упр. 9.5**] На примере факториала, раз:

si/libexp.scm

```
(define (fact2 n)
  (if (= n 0) 1
      (* n (fact2 (- n 1))) ) )

(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0))
      (fact2 n)
      `(fact1 ,n) ) )
```

и два:

si/librun.scm

```
(define (fact1 n)
  (if (= n 0) 1
      (* n (fact1 (- n 1))) ) )
```

Во время предобработки директивы анализируются, и компилятор делает заметку, что для раскрытия макроса `factorial` ему потребуется загрузить библиотеку `si/libexp`, а к готовому приложению должна быть подключена `si/librun`. Таким образом, ресурсы используются только тогда, когда они действительно необходимы (дальнейшее развитие идеи см. в [DPS94b]). Итог подведён на рисунке 9.3: сначала собираются необходимые библиотеки, затем программа компилируется и связывается с макробиблиотекой времени исполнения, а библиотека времени раскрытия выбрасывается за ненадобностью.

9.6.2. Единый мир

После предыдущего раздела вам может показаться, что во всём виновата множественность миров и достаточно просто их объединить. Как бы не так!

В едином мире выражения считываются, раскрываются, подготавливаются и исполняются в общей памяти. Следовательно, состояние макроэкспандера переплетается с состоянием исполнителя кода. Если применяется эндогенный подход, то макровычислитель (функцию `macro-eval`) логично будет отождествить с обычным вычислителем `eval` (естественно, во всех его вариациях из восьмой главы). [**см. стр. 325**] Так как мир един, то раскрываемые макросы имеют неограниченный доступ ко всему его содержимому. Верно и обратное, так что здесь нет особого различия между библиотеками времени исполнения и раскрытия. Следующий код в едином мире ведёт себя именно так, как ожидается:

```
(define (fact n)
  (if (= n 0) 1
      (* n (fact (- n 1))) ) )
```

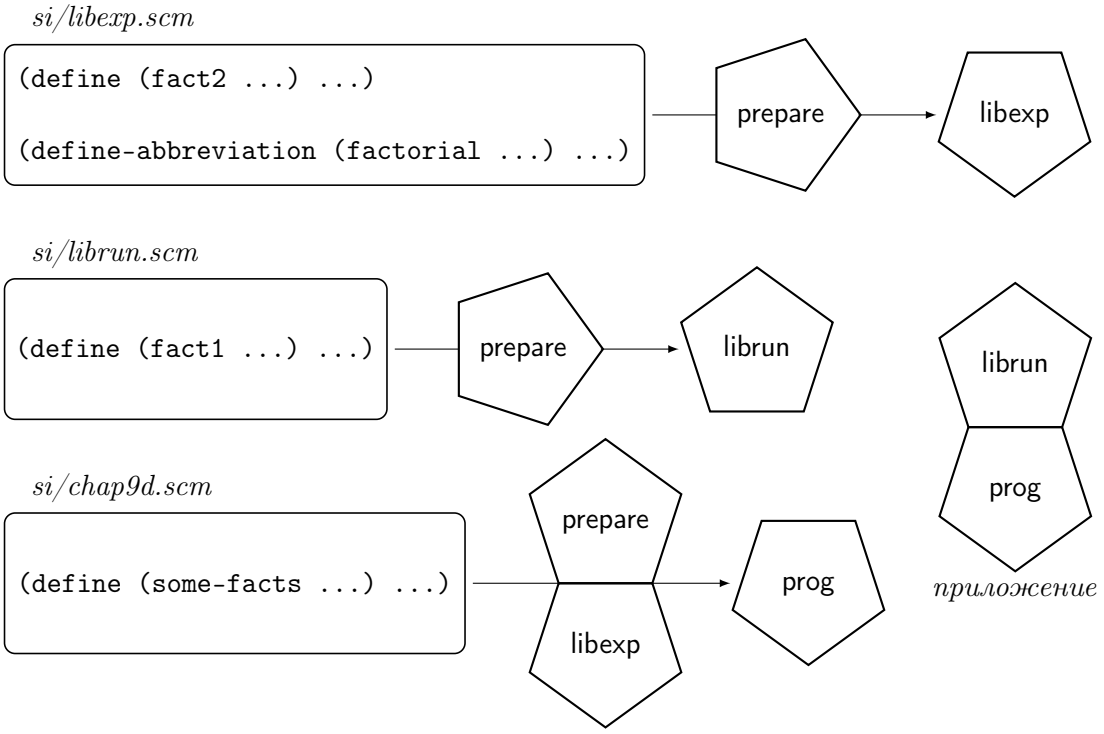


Рис. 9.3. Сборка приложения во множественных мирах.

```
(define-abbreviation (factorial n)
  (if (and (integer? n) (> n 0))
      (fact n)
      `(fact ,n) ) )
```

Но есть и ложка дёгтя в бочке мёда: с таким подходом гораздо сложнее передавать (и продавать) программы, так как трудно, а порой и вовсе невозможно выделить в программе необходимые для её работы части. Следовательно, если мы хотим передать программу кому-то другому, то вынуждены передавать её целиком и полностью, чтобы ничего не забыть. Конечно, существуют специальные анализаторы зависимостей, называемые *tree-shakers*, которые могут помочь вытрясти из программ всё ненужное. Но чаще всего, как видно из множества примеров, доступных в Интернете, люди воздерживаются от включения экспортируемых макросов в готовые приложения: код предоставляется либо полностью раскрытым, либо же использует макросы только локально, внутри себя. В любом случае конечный пользователь оказывается лишён всей их выразительной силы.

Если вы не доверяете алгоритмам определения зависимостей, то ничего не остаётся, кроме как самостоятельно выбирать, что включать в программу, а что нет. Всего имеется три типа ресурсов: необходимые исключительно для

раскрытия макросов, необходимые только при исполнении программ и (самое сложное) необходимые на обеих этапах.

За подготовку программ к исполнению отвечает функция `compile-file`. Единые миры по отношению к ней разделяются на две группы, одну из которых можно назвать *истинно едиными мирами*. Предмет спора: макросы, доступные `compile-file` во время предобработки. Кроме того, даже в истинно едином мире возможны разногласия, что даёт в итоге целых три вариации исходной идеи:

1) В истинно едином мире макросы вездесущи, доступны всегда и отовсюду. Следовательно, компилируемый на лету модуль вполне может пользоваться макросами «родителя». Данная ситуация показана на рисунке 9.4, где макрос `factorial`, определяемый интерактивно, используется при компиляции другого модуля. Вопрос: а что с макросами, которые определяются внутри компилируемого модуля?

   а) С ними следует обращаться на общих началах: раз мир един, то они должны быть доступны внешнему коду. Создаваемые внутри макросы просто добавляются в глобальное общее пространство макросов и остаются там после компиляции подмодуля. Этот случай показан на рисунке 9.5 где `(factorial 5)` возвращает `120`.

   б) В другом случае глобальность понимается на уровне отдельных модулей: определяемые макросы видны только в пределах своего модуля. Глобальные макросы разделяются на *локально глобальные*, определяемые в этом же модуле, и *сверхглобальные*, определяемые вне его, но видимые благодаря иерархической компиляции модулей. Для программы на рисунке 9.5 этому случаю соответствует возвращаемая `(factorial 5)` ошибка.

2) Наконец, если мир не истинно един, то модуль может пользоваться исключено самостоятельно определяемыми макросами. В этом случае есть отдельное пространство макросов для интерактивной сессии и отдельные, персональные и изначально пустые пространства для компилируемых модулей. Однако, несмотря на то, что пространства макросов отделены друг от друга, память остаётся общей и всем макросам во время раскрытия доступны глобальные разделяемые ресурсы. Например, на рисунке 9.5 внутренний макрос `factorial` вполне может использовать функцию `fact`, определённую в интерактивной сессии.

После внимательного изучения рисунков 9.4 и 9.5 можно прийти к выводу, что в обоих случаях функция `fact` определяется глобально в интерактивной сессии, хотя похоже, что она нужна только для раскрытия макроса `factorial`. Можно было бы определить её локально, как это сделано в `si/chap9b.scm`. [**см. стр. 384**]

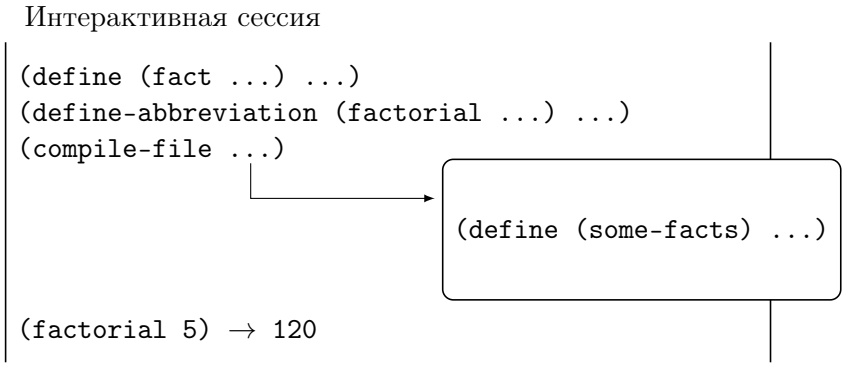


Рис. 9.4. Истинно единый мир.

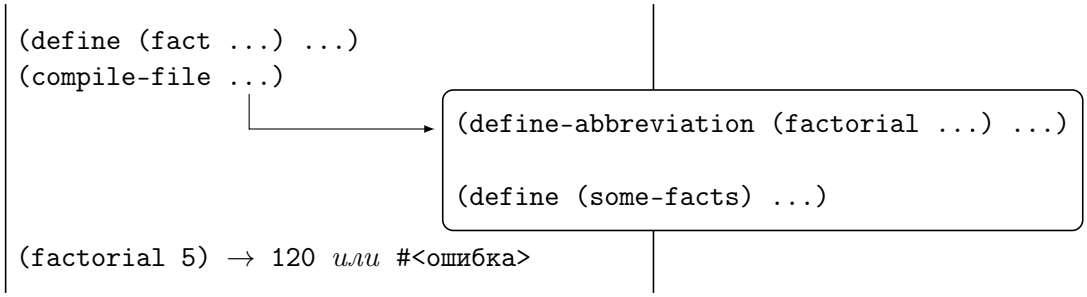


Рис. 9.5. Истинно единый мир с эндогенной компиляцией.

Но что если `fact` пригодилась бы не только этому макросу? Беда в том, что макроэкспандер воспринимает только определения макросов. Если для раскрытия потребуются глобальные переменные, вспомогательные функции или другие вещи, то он просто не поймёт, о чём идёт речь. Если во множественных мирах при экзогенном подходе с этим нет вообще никаких проблем, то в едином мире или при эндогенном задании макросов придётся немного потрудиться. Вспомните, что сама суть этих вариаций в том, что во время раскрытия макросов происходят некоторые вычисления. Естественно, выполняемые вычисления вовсе не обязаны быть прямо связанными с раскрытием макросов. В Common Lisp и некоторых реализациях Scheme существует соответствующая форма, называемая `eval-when`, способная просто вычислить что-нибудь, не подставляя полученный результат в раскрываемый код. (Как обычно, чтобы имена не пересекались, её местный аналог будет называться по-другому: `eval-in-abbreviation-world`.)

Форма `eval-in-abbreviation-world` позволяет определять вспомогательные функции, константы, проводить любые необходимые вычисления во время раскрытия макросов и исключительно ради него. Фактически, эта форма явно отделяет некоторые выражения от внешнего мира вычислений. Взгляните на рисунок 9.6: функция `fact`, определяемая `eval-in-abbreviation-world`, гарантированно может быть использована для раскрытия `factorial` внутри

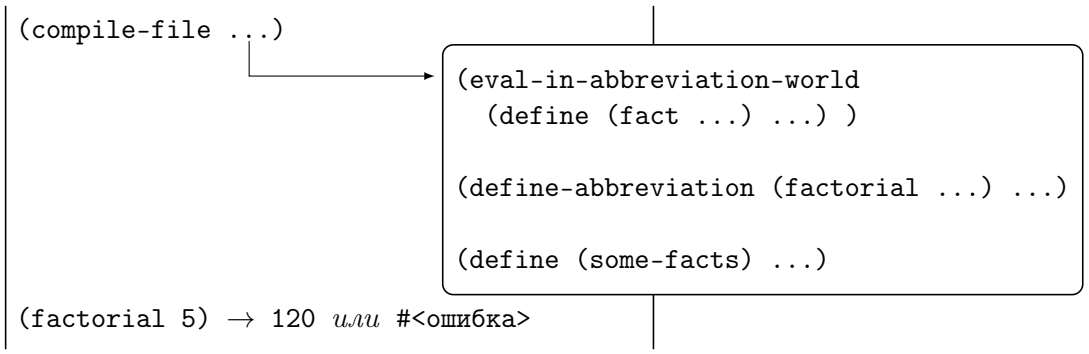


Рис. 9.6. Явные эндогенные вычисления.

определения `some-facts`. Возможность использовать `factorial` и/или `fact` по возвращении в интерактивную сессию зависит уже от «герметичности» мира макросов.

Для реализации такого поведения достаточно научить макроэкспандер реагировать всего лишь на одно ключевое слово `eval-in-abbreviation-world`, вынуждающее его отправлять `cdr` этой формы соответствующему вычислителю — тому самому, который в функции `naive-endogenous-macroexpander` назывался `macro-eval`.

```
(define (expand-expression exp macroenv)
  (define (evaluation? exp)
    (and (pair? exp)
         (eq? (car exp) 'eval-in-abbreviation-world) ) )

  (define (macro-call? exp)
    (and (pair? exp) (find-expander (car exp) macroenv)) )

  (define (expand exp)
    (cond ((evaluation? exp) (macro-eval `(begin . ,(cdr exp))))
          ((macro-call? exp) (expand-macro-call exp macroenv))
          ((pair? exp)
           (let ((newcar (expand (car exp))))
             (cons newcar (expand (cdr exp))) ) )
          (else exp) ) )

  (expand exp) )
```

Функция `macro-eval` — это именно тот вычислитель, который реализует внутренний макроязык. Это может быть и обычная функция `eval`, но использующая чистое глобальное окружение, отдельное от того, которым пользуется интерактивная сессия. В таком случае и `(factorial 5)`, и `(fact 5)` вернут ошибку.

Формы `eval-in-abbreviation-world` понимаются как непосредственно исполнимые директивы для макроэкспандера, включённые в программу. Они, естественно, выполняются во время раскрытия макросов, поэтому вот так писать нельзя:

```
(let ((x 33))
  (eval-in-abbreviation-world (display x)) )
```

Если в процессе раскрытия макросов потребуется создавать локальные переменные, — что для формы `eval-in-abbreviation-world` невозможно, так как она фактически является лишь вызовом `eval` на глобальном уровне, — то следует использовать форму `compiler-let`, присутствующую в Common Lisp I [Ste84] (но не в Common Lisp II [Ste90]). Необходимо лишь научить `expand-expression` распознавать её.

Во время раскрытия макросов ещё нет никакого окружения исполнения программы. Верно и обратное: во время исполнения уже нет окружения раскрытия. Поэтому вот так использовать макросы тоже нельзя:

```
(define-abbreviation (foo ...) ...)
(apply foo ...)
```

Форма `eval-in-abbreviation-world` снова позволяет определить `define-abbreviation` как элементарный макрос:

```
(define-abbreviation (define-abbreviation call . body)
  `(eval-in-abbreviation-world
     (install-macro! ',(car call) (lambda ,(cdr call) . ,body))
     #t ) )
```

Главной особенностью данного определения является возможность размещать макроопределения где угодно, а не только на глобальном уровне. Следовательно, такая программа вполне допустима:

```
(begin (define-abbreviation (foo x) ...)
       (bar (foo 34)) )
```

Хотя, конечно, чересчур сильная зависимость от порядка раскрытия макросов считается плохим вкусом, равно как и извращения вроде

```
(if (bar) (begin (define-abbreviation (foo x) ...)
                 (hux) )
          (foo 35) )
```

Большинство макросистем отличают макросы, объявляемые глобально, от других, подобно тому, как глобальные формы `define` отличаются от внутренних. Также глобальные макроопределения гарантированно обрабатываются последовательно сверху вниз. Это позволяет, например, сначала определить класс, а только потом его наследников.

Стоит отметить, что в предыдущих примерах возможны определённые нестыковки, касающиеся вычислителей, обрабатывающих выражения. Если

эти вычислители различны, то у них могут возникнуть проблемы с ошибочным использованием структур данных друг друга, подобные тем, которые имеются у рефлексивных интерпретаторов. [**см. стр. 325**] Форма `install-macro!` из определения `define-abbreviation` обрабатывается `macro-eval`, она должна изменить текущий набора макросов, доступных `find-expander`. Но ведь функция `find-expander` будет вызываться `eval`, а не `macro-eval`! Ладно, оставим эти затруднения с совместимостью на потом.

Одной из весьма полезных возможностей `eval-in-abbreviation-world` является создание глобальных переменных для макроэкспандера. Часто макросы применяются для того, чтобы абстрагироваться от некоторых особенностей реализации. Например, в старых версиях Scheme функция `apply` была бинарной, а не n-арной, как сейчас. Легко представить макрос, который бы раскрывался тем или иным образом, чтобы скрыть данное различие. Помимо исправления уже существующих недостатков, макросы можно использовать и для предотвращения новых проблем, заранее выполняя раскрытие правильным образом. Так часто поступают в сложных системах вроде MEROON для обеспечения переносимости. Очевидно, в обоих случаях необходимо иметь в распоряжении этот самый список особенностей реализации, куда можно подглядывать во время раскрытия макросов на предмет интересующих деталей. Например, как-то так:

```
(define-abbreviation (apply-foo x y z)
  (if (memq 'binary-apply *features*)
      `(apply foo (cons x (cons y z)))
      `(apply foo x y z) ) )
```

Переменная `*features*` должна быть доступна экспандеру. Легче всего её создать с помощью `eval-in-abbreviation-world`, передав ей соответствующую форму `define`. Эта форма будет вычислена на глобальном уровне в мире макросов, что приведёт к созданию необходимой переменной. Это может делаться, например, в каком-нибудь автоматически загружаемом конфигурационном файле:

```
(eval-in-abbreviation-world
  (define *features* '(31bit-fixnum binary-apply)) )
```

9.6.3. Совместные вычисления

Ещё одной часто реализуемой вариацией единого мира является одновременное выполнение предобработки и вычислений. Выражения считываются, раскрываются и незамедлительно вычисляются. В коде это выглядит так:

```
(define (simultaneous-eval-macroexpander exps)
  (define (macro-definition? exp)
    (and (pair? exp)
         (eq? (car exp) 'define-abbreviation) ) )
```

```
(if (pair? exps)
    (if (macro-definition? (car exps))
        (let* ((def       (car exps))
               (name      (car (cadr def)))
               (variables (cdr (cadr def)))
               (body      (cddr def)) )
          (install-macro!
           name (macro-eval `(lambda ,variables . ,body)) )
          (simultaneous-eval-macroexpander (cdr exp)) )
        (let ((e (expand-expression (car exps) *macros*)))
          (eval e)
          (cons e (simultaneous-eval-macroexpander (cdr exps))) ) )
    '() ) )
```

Обратите внимание на вычислители: `eval` отвечает за вычисление готовых выражений целевого языка, а `macro-eval` за вычисление выражений макроязыка. Они разделены, так как выполняют различную работу на различных этапах.

9.6.4. Переопределение макросов

Восклицательный знак в имени функции `install-macro!` подразумевает возможность переопределения макросов: если состояние макроэкспандера можно модифицировать, добавляя новые макросы, то точно таким же путём можно изменять определения уже существующих. Это может оказаться полезным, например, при отладке с помощью интерпретатора: определяем функцию, тестирующую отлаживаемый макрос, а затем вызываем её после каждого исправления. Но чтобы такой подход сработал, интерпретатор должен честно раскрывать макросы в последний момент, непосредственно перед вызовом функции. Однако чаще всего макросы раскрываются лишь один раз в самом начале, строго разделяя процессы раскрытия и исполнения. Очевидно, что однажды раскрытый макрос больше не зависит от своего определения. Изменив определение, мы повлияем лишь на дальнейшие раскрытия этого макроса. Таким образом, можно сказать, что определения макросов *гиперстатичны*. [**см. стр. 78**]

9.6.5. Итоговое сравнение

Одна из главных проблем с макросами состоит в том, что нельзя просто так взять и изменить макросистему, предоставляемую реализацией. Вероятно, подобное ограничение на свободу экспериментирования отчасти ответственно за современную ситуацию с макросами в Лиспе и Scheme.

Системы с множественными мирами и экзогенными макроопределениями, кажется, более точно выражают идею, но они довольно слабо распространены.

Эндогенный подход во множественных мирах есть лишь частный случай экзогенного, где вне обрабатываемой программы определяются примитивы вроде `define-abbreviation` и т. д., с помощью которых реализуются пользовательские макросы. Наконец, единые миры являются наиболее распространённым вариантом, хотя и вызывают определённые затруднения с передачей готового программного обеспечения. Данное сравнение, однако, слегка поверхностно, так как не учитывает двух важных аспектов: компиляции макросов и определения макросов с помощью макросов. О них мы сейчас и поговорим.

Компиляция макросов

До сих пор мы неявно предполагали, что раскрытием макросов занимается интерпретатор. Очевидно, это сказывается на производительности, особенно если экспандер выполняет сложные вычисления (что редко, но бывает). Множественные миры и экзогенное определение макросов [**см. стр. 374**] позволяют на лету сгенерировать специализированный компилятор, понимающий все необходимые макросы, что должно благотворно сказаться на быстродействии. При эндогенном подходе к описанию макросов одним из решений может быть создание компилирующего аналога функции `macro-eval`. В едином мире для этого предназначена пара функций `compile-file` и `load`. Вся проблема в том, как скомпилировать *определения* макросов.

Это отнюдь не тривиальная задача! Наши текущие экспандеры просматривают раскрываемые выражения на предмет форм `define-abbreviation` и определяют соответствующие макросы на лету. Но теперь, при компиляции, они должны не только запомнить определения где-то у себя внутри, но и оставить в программе нечто, описывающее просмотренные определения. Если `define-abbreviation` является на самом деле макросом, раскрывающимся в форму `install-macro!`, то исполняемая программа сама всё сделает. Если же `define-abbreviation` и компания — это лишь синтаксические маркеры для экспандера, то всё придётся делать компилятору: откомпилировать тело макроса в функцию-экспандер, которая будет динамически вычислять, загружать и исполнять результаты раскрытия. Однако, при таком подходе возникают некоторые неоднозначности: к примеру, как тогда понимать `(eval-in-abbreviation-world (load file))`? Очевидно, здесь имеется в виду, что *компилятор* должен динамически загрузить файл. Но функция `load` — это же обёртка над `eval`, что подразумевает динамические вычисления, вычисления во время исполнения программы. Та же самая ситуация возникает и при явном обращении к `eval`: `(eval-in-abbreviation-world (eval '(define-abbreviation ...)))`. В обоих случаях вместо `eval` должна вызываться `macro-eval`, чтобы эти выражения имели смысл, так как *исполняемой программой* сейчас является сам компилятор. Таким образом, в макромире действительно используется собственный язык с собственной `eval`, которая идентична `macro-eval` целевого языка.

Всё это наталкивает нас на мысль, что для определения макросов необходимо использовать другие макросы, а не просто синтаксические маркеры, на которые реагирует экспандер-препроцессор. Если макросы требуется сохранять в обработанных программах, то они вынуждены будут стать полноценными объектами целевого языка. В таком случае анализ определений макросов и установка обработанных определений в макроэкспандер тоже должны быть разделены, так как они принадлежат различным мирам. Чуть позже мы рассмотрим этот вопрос подробнее. [**см. стр. 400**]

Макросы, определяющие макросы

Время от времени бывает удобным определить макрос, который бы создавал другие макросы. [**см. стр. 403**] Это не бред, а вполне естественное желание максимально полно использовать выразительные возможности, предоставляемые макросами. Например, пусть мы разрабатываем объектную систему. Логичным будет дать пользователю возможность создать класс `Point` таким образом, чтобы его аксессоры `Point-x` и `Point-y` были на самом деле макросами, а не функциями. [**см. стр. 502**] В таком случае форма `define-class` вынуждена будет сама создать эти новые макросы.

Рассмотрим для примера макрос `define-alias`, делающий свой первый аргумент тождественным второму. Вот два эквивалентных варианта его реализации:

```
(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     `(,',oldname . ,parameters) ) )

(define-abbreviation (define-alias newname oldname)
  `(define-abbreviation (,newname . parameters)
     (cons ',oldname parameters) ) )
```

И снова, если `define-abbreviation` это макрос, то при рекурсивном раскрытии определения `define-alias` макроэкспандер раскроет и вложенное определение. А вот если бы `define-abbreviation` была синтаксическим маркером, то не факт: вдруг макроэкспандер выполняет работу за два прохода и на втором его уже не интересуют всякие маркеры, он только раскрывает макросы, а на всё остальное даже не смотрит. Таким образом, далее мы будем считать `define-abbreviation` именно предопределённым макросом; скорее даже примитивным, так как его нельзя определить, не будь он изначально встроен в реализацию.

9.7. Область видимости макросов

Если область видимости локальных макросов, определяемых формами наподобие `let-syntax` и `letrec-syntax`, вполне очевидна, то вот определения

`define-syntax` не настолько однозначны, потому что глобальные макросы можно использовать по-разному. Рассмотрим возможные варианты подробнее на примере Meroon — переносимой объектной системы для Scheme, допускающей как интерпретацию, так и компиляцию. Ядро Meroon называется Meroonet. (См. одиннадцатую главу.) В этой системе используются три типа макросов:

1) **Случайные макросы.** Такие макросы определяются, тут же используются, а дальше их можно выбросить. В принципе, для них прекрасно подходят формы `let-syntax` или `macrolet`, если, конечно, они поддерживаются реализацией. В качестве примера возьмём функцию `make-fix-maker` из Meroonet. Она создаёт «треугольник» замыканий вида `(lambda (a b c ...) (vector cn a b c ...))`, но её определение записано вручную. [**см. стр. 515**] А можно было бы написать для этого специальный макрос:

```
(define-abbreviation (generate-vector-of-fix-makers n)
  (let* ((numbers (iota 0 n))
         (variables (map (lambda (i) (gensym)) numbers)) )
    `(case size
       ,@(map (lambda (i)
                (let ((vars (list-tail variables (- n i))))
                  `((,i) (lambda ,vars (vector cn . ,vars))) ) )
              numbers )
       (else #f) ) ) )
```

Здесь важно то, что подобные макросы «одноразовые». У них очень ограниченная область видимости: в любом случае не шире модуля, где они определены.

2) **Локальные макросы.** Исходный код Meroon разделён на 25 небольших файлов. Естественно, внутри этих файлов используется некоторое количество макросов для собственных нужд, например, `when`. Область видимости этих макросов должна быть ограничена исключительно файлами Meroon. Макросы, локальные для объектной системы, не имеют права загрязнять внешний мир.

3) **Экспортируемые макросы.** Meroon экспортирует три своих макроса: `define-class`, `define-generic` и `define-method`. Эти макросы существуют в первую очередь для пользователей Meroon, но и для самой системы они тоже могут быть полезны. Очевидно, макросы вроде `when` запрещено использовать в макросах третьего типа, так как иначе `when` выйдет за пределы своей системно-локальной области видимости. Целевой язык экспортируемых макросов — это язык пользователя, а не Meroon.

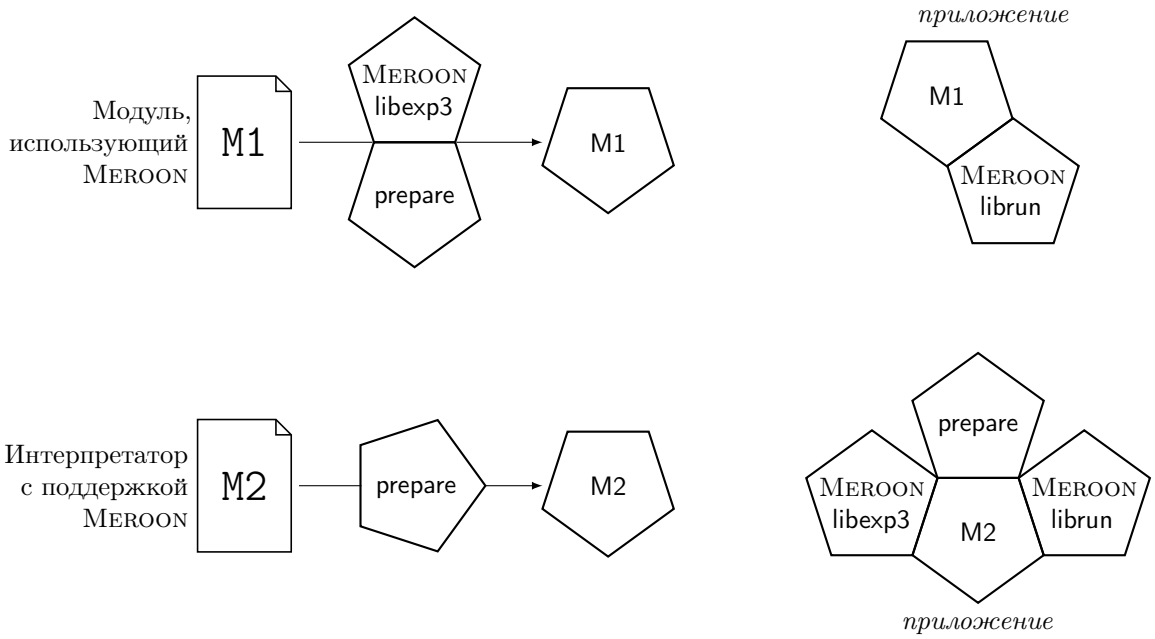


Рис. 9.7. Варианты использования макросов.

Существует не только три типа макросов, но и три способа их использования, показанные на рисунке 9.7.

1) **Для компиляции Meroon.** Для этого необходимо раскрыть все макросы, присутствующие в исходном коде системы. Естественно, после этого не остаётся ни одного макровызова, что снимает все проблемы. Но от макросов третьего типа должно оставаться нечто, что позволяет их использовать снаружи, ведь они всё же экспортируются.

2) **Для компиляции программ, использующих Meroon.** В таком случае достаточно раскрыть лишь макросы Meroon третьего типа, встречающиеся в пользовательском коде.

   Очевидно, что для этого в экспандер компилятора необходимо установить соответствующие определения. Иными словами, подключить библиотеку времени раскрытия макросов Meroon третьего типа. Заметьте, что библиотеки для макросов второго и первого типов при этом не нужны.

3) **Для создания интерпретатора с поддержкой Meroon.** Получаемая интерактивная сессия должна быть способна на лету раскрывать макросы Meroon третьего типа. Соответственно, эти макросы вместе со своей библиотекой времени раскрытия должны быть подключены к макроэкспандеру самого интерпретатора.

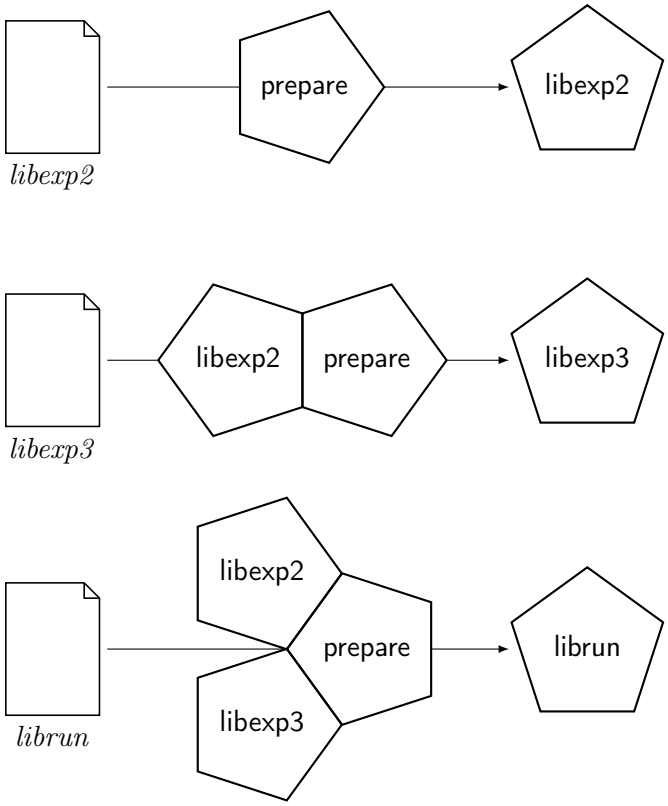


Рис. 9.8. Раскрутка Meroon.

На рисунке 9.8 показан один из способов сборки модулей Meroon, известный как *раскрутка* (bootstrapping). Библиотека раскрытия макросов второго типа собирается исключительно как вспомогательная при сборке остальной части объектной системы. Всё это напоминает так называемые T-диаграммы [ES70], отображающие связи между языками реализации, реализуемыми языками и целевыми языками применительно к процессу компиляции.

После рассмотрения различных типов макросов и вариантов их использования можно сделать вывод, что для подобных систем отлично подходит идея множественных миров, так как в едином мире обычно отсутствуют ограничения на область видимости. Одним из способов организации макросов являются *пакеты* (как, например, в Common Lisp или ILOG Talk). Внутренние макросы можно просто собрать в пакет, получив таким образом для них отдельное пространство имён, которое не будет пересекаться с пользовательскими. В едином же мире все три типа макросов, которые мы так тщательно разделяли, оказываются сваленными в одну кучу.

Meroonet, ядро объектной системы Meroon, рассматривается подробнее в одиннадцатой главе. Она определяет всего три макроса, но сама их при этом не использует, избегая таким образом вышеупомянутых проблем. Данные макросы относятся к третьему типу по нашей классификации и опреде-

ляются с помощью специального макроса `define-meroonet-macro`. Сделано так потому, что экспортируемые макросы связаны с пользовательской макросистемой, от которой MEROONET следует абстрагироваться. Нельзя ожидать, что ей будет доступна функция `install-macro!` или что-либо другое, поэтому она вводит интерфейс в виде макроса `define-meroonet-macro`, который должен быть реализован с помощью примитивов конкретной макросистемы. Допустим, таким примитивом является `define-abbreviation`. Учитывая требование экспортируемости макросов — возможности использовать их как в модуле, где они определяются, так и в модулях, которые динамически загрузят модуль с определениями, — `define-meroonet-macro` можно реализовать следующим образом:

```
(define-abbreviation (define-meroonet-macro call . body)
  `(begin (define-abbreviation ,call . ,body)
          (eval '(define-abbreviation ,call . ,body)) ) )
```

9.8. Вычисления и макрораскрытие

В интерактивной сессии раскрытие макросов и вычисление выражений очень тесно связаны друг с другом, за раскрытием всегда следует вычисление. В данном разделе мы внимательнее присмотримся к этой неразлучной паре.

Макрораскрытие — это первый этап подготовки программы к исполнению. После подготовки можно вычислять выражения, составляющие программу. Функция `eval` занимается исключительно вычислениями, она знает только чистый язык, лишённый всяких макросов. Для динамического вычисления выражений, содержащих макросы, их необходимо явно пропускать через экспандер перед передачей `eval`. Соответственно, вполне возможен вариант, когда в сессии одновременно существуют два независимых макроэкспандера. Правда, такая практика обладает сомнительной полезностью; обычно всё же в пределах одного языка есть только один вычислитель и один макроэкспандер, так как иначе возникли бы определённые трудности со стандартными макросами вроде `cond`, `let` или `case`. Поэтому обычно `eval` на самом деле является чем-то вроде `(lambda (e) (pure-eval (macroexpand e *macros*)))`, где `pure-eval` — это настоящий вычислитель чистого языка, `macroexpand` — явная функция-макроэкспандер, а `*macros*` — текущий список макросов, известных интерпретатору.

Сказанное сейчас про `eval` в равной мере касается и `macro-eval`: у неё тоже есть своя переменная `*macros*`, которая хранит макросы, необходимые во время подготовки. Вопросов эта пара из вычислителя и экспандера вызывает немало. Например, вспомним [**см. стр. 332**] о существовании `eval/ce`, которая позволяет проводить вычисления в текущем окружении. Эта специальная форма захватывает всё видимое ей лексическое окружение. По идее, `eval/ce` должна уметь обращаться с выражениями, содержащими макросы,

но значит ли это, что она захватывает окружающие её макроопределения? Другими словами, если написать

```
(let-syntax ((foo ...))
  (eval/ce (read)) )
```

то раскроет ли `eval/ce` макровызовы `foo` в считанном выражении? Если да, то она должна сохранять не только весь лексический контекст, но также и макроконтекст — полное состояние макроэкспандера; только так `eval/ce` сможет корректно раскрыть `foo`. Поскольку подобные приёмы серьёзно нарушает принцип независимости процессов раскрытия и исполнения, они считаются плохим стилем. Вообще говоря, `eval` тоже по-хорошему должна быть чистой и не вызывать всяческих раскрытий, когда её не просят, но это неудобно, непрактично и т. д., и т. п.

После определения макроса его можно использовать при последующих раскрытиях. Предположим, ранее мы определили весьма полезный макрос `when`. Будет ли работать следующее определение?

```
(define-abbreviation (whenever condition . action)
  (when condition (display '(whenever is called)))
  `(when ,condition . ,action) )
```

В этом макросе `when` используется дважды. Второй раз приходится на результат раскрытия, так что здесь проблем нет. Но вот в первый раз `when` используется во время раскрытия определения макроса, то есть когда вычислениями занимается совершенно другой вычислитель, который может вообще не знать про `when`: ведь макросы-то сейчас раскрываются в обрабатываемой программе, а не в макроопределениях! Макрос `when`, доступный в том числе во время самого процесса раскрытия, должен определяться вот так:

```
(eval-in-abbreviation-world
  (define-abbreviation (when condition . body)
    `(if ,condition (begin . ,body)) ) )
```

Похожая ситуация возникает и в выражении

```
(define-abbreviation (foo)
  (define-abbreviation (bar)
    (when ...)
    (wrek) )
  (hux) )
```

Здесь внутреннее определение `bar` вводится для расширения макроязыка, а не пользовательского. Макрос `bar` определяется, чтобы облегчить запись определения другого макроса. Однако, даже если семантика данного выражения сомнений не вызывает, то его прагматика* слегка неясна, так как здесь обра-

* Прагматика — понятие лингвистики: смысл выражения, определяемый его контекстом. Для языков программирования таким контекстом является компьютер, исполняющий программы. То есть прагматика — это принципы реализации языка для конкретных вычислительных машин, тогда как семантика — это просто абстрактный смысл выражений языка.

зуется своеобразный порочный круг. Ведь различными могут быть не только целевой и макроязыки, но и язык для описания описаний макросов может отличаться от макроязыка, и так далее. Другими словами, нельзя быть уверенным в том, что форма `when` в описании внутреннего макроса `bar` будет понята верно. Если взять действительно различные языки, проблема станет более явной. Например, макросы в программах на Си++ раскрываются препроцессором Си++. Представьте, что эти макроопределения генерируются программой на Perl. Очевидно, что она должна возвращать эти определения именно на Си++, так как программа на Perl для компилятора Си++ — это одна большая синтаксическая ошибка, несмотря на то, что в ней и есть некий подразумеваемый смысл с точки зрения Perl.

И что же делать? Одним из вариантов является строгое следование семантике — то есть поддержка всех используемых языков. К счастью, их редко когда требуется больше двух. Но тогда возникают проблемы многоуровневой интерпретации, рассмотренные ранее на примере рефлексивных интерпретаторов. [**см. стр. 361**]

Ещё одна возможность заключается в объединении всех языков, используемых макроэкспандером. То есть при раскрытии макросов все вычисления выполняются в едином мире с одним-единственным языком описания макросов. Но тогда все наши усилия по разделению языков провалятся в тартарары.

Третьим вариантом — именно его использует R^5RS — будет ограничение языка описания макросов. Если его выразительные возможности сводятся лишь к перестановкам и повторениям уже написанных конструкций целевого языка, то такие макросы не смогут создавать другие макросы и исходная проблема не возникает в принципе.

Короче говоря, снова мы приходим к тому, что очень важно различать используемые языки и стараться их не смешивать. Если используемая реализация Scheme поддерживает многопоточность и распределённые вычисления, то отсюда вовсе не следует, что её макроэкспандер тоже это умеет. Например, если в макромире есть продолжения, то они не обязательно ведут себя так же, как и в Scheme. Не надо, дорвавшись до них, тут же бежать показывать своё мастерство с помощью чего-то подобного:

```
(define-abbreviation (foo x)
  (call/cc (lambda (k)
             (set! the-k k)
             x )) )
(define-abbreviation (bar y)
  (the-k y) )
```

9.9. Применения макросов

В данном разделе рассматриваются типичные варианты использования макросов. Предназначением макросов является, конечно же, преобразование

программ, но существует множество причин, по которым эти преобразования выполняются. Среди них можно выделить следующие:

- *Сокращение* объёма набираемого кода (особенно во время интерактивной работы или отладки). Если мы хотим писать `(trace foo)`, чтобы получать в дальнейшем сообщения обо всех вызовах функции `foo`, то здесь потребуется макрос, дабы значение `foo` не вычислялось при подобном вызове.

- *Облагораживание* кода для сокрытия некрасивого синтаксиса. Например, использование `bind-exit` вместо `call/ep`, чтобы не писать каждый раз эту надоедливую `lambda`.

- *Абстрагирование*, позволяющее скрывать детали реализации, — главная задача макросов. Не следует выставлять напоказ то, что может в скором времени измениться, или нечто личное, куда не должны совать свой нос посторонние. Примерами могут быть `define-class` MEROONET или `syntax-rules` Scheme.

Отдельно стоит упомянуть ещё один важный подвид макросов, использующихся для обеспечения переносимости. Вроде упомянутого ранее `apply-foo` [**см. стр. 393**], который был применён для скрытия истинной арности `apply`. Или, например, другая проблема: до R^{4}RS пустой список `()` не обязательно имел булево значение `#t`. Ожидаемое поведение можно обеспечить, используя везде специальный макрос, скажем, `meroon-if`, который определяется следующим образом:

```
(define-abbreviation (meroon-if condition consequent . alternant)
  `(if (let ((tmp ,condition))
         (or tmp (null? tmp)) )
       ,consequent . ,alternant ) )
```

Естественно, подобных уловок можно избежать, если изначально правильно писать программы. Но, увы, не все программы хорошо написаны, а среди «плохих» вполне могут оказаться и весьма полезные. Подобные макросы являются одним из способов быстрого портирования необходимого программного обеспечения.

Ещё одним применением макросов является гарантирование некоторых оптимизаций независимо от используемого компилятора. Одной из таких оптимизаций является *инлайнинг* — прямая подстановка кода функций вместо их вызова. Некоторые реализации предоставляют специальные директивы для этого, но так как подобный способ не является переносимым, а также не всегда позволяет использовать подобные функции вне модуля, где они определены, нам остаётся лишь реализовать всё самостоятельно. Для этого достаточно связать необходимую функцию с одноимённым макросом:

```
(define-abbreviation (define-inline call . body)
  (let ((name      (car call))
        (variables (cdr call)) )
    `(begin
       (define-abbreviation (,name . arguments)
         (cons (cons 'lambda (cons ',variables ',body))
               arguments ) )
       (define ,call (,name . ,variables)) ) ) )
```

Правда, этот макрос тоже не абсолютно переносим, так как в некоторых диалектах запрещено одновременно иметь макрос и функцию с одинаковым именем. Другие диалекты более снисходительны, но в них второе определение (функция) полностью заменит первое (макрос), так что никакого инлайнинга не будет. Кроме того, при таком подходе чрезвычайно сложно создать рекурсивную встраиваемую функцию, не зациклив при этом макроэкспандер (хотя некоторые с этим не согласны, см. [Bak92b]). Наконец, стоит избегать использования встраиваемых функций как значений (например, не передавать их в `apply`), потому как это позволит компилятору вообще не создавать соответствующую полноценную функцию.

Во множественных мирах данный макрос работает прекрасно. Макроэкспандер раскрывает все вызовы определённой таким образом функции, являющиеся на самом деле макровызовами одноимённого макроса. Вместо них экспандер подставляет код функции (в том числе в форме `define`, которая определяет «настоящую» функцию). В результате, после раскрытия макросов все прямые вызовы функции оказываются заменены кодом, а остальные упоминания имени являются обычными ссылками на обычную функцию — как раз это нам и надо.

9.9.1. Иные качества

Макросы Scheme стандарта R^{5}RS имеют четыре важных особенности:

1) они гигиеничны (подробнее в разделе 9.10);

2) они определяются с помощью *паттернов*;

3) они раскрываются простыми подстановками;

4) они не имеют внутреннего состояния.

Преимуществом определения синтаксиса макросов в виде паттернов является возможность тонкого контроля над требуемой формой макровызовов. Более того, не требуется писать ни одной строчки кода для обеспечения этого контроля. Например, многие макросы последним аргументом принимают список, являющийся неявным вызовом `begin`. Им не требуется проверять, что этот список является корректным (оканчивается на `()`, а не точечную пару) —

эта проверка автоматически выполняется макроэкспандером, который расскажет обо всех встреченных им синтаксических ошибках, включая и эту. Для реализации подобных проверок существуют эффективные алгоритмы сопоставления с образцом (pattern matching), например, рассмотренные в работах [Que90b, QG92, WC94].

Подстановки также реализуются квазицитированием, но такой подход позволяют выражать лишь операции над списками и векторами. В частности, здесь нет даже четырёх арифметических действий и самих чисел вообще. [**см. упр. 9.2**]

Отсутствие у макросов собственного состояния — более серьёзное неудобство. Макросы в таком случае являются контекстно-независимыми, что с одной стороны, конечно, удобно, но вызывает некоторые сложности с определением макросов вроде `define-class`, которому серьёзно облегчит жизнь поддерживаемая в адекватном состоянии иерархия ранее определённых классов. Ведь для создания классов-наследников необходимо иметь информацию об их родителях: к примеру, количество и имена их полей.

Иногда действительно хочется иметь макросы, учитывающие контекст своего раскрытия. Представьте себе макрос `date`, раскрывающийся в текущую дату. Он бы пригодился, например, в качестве простой системы управления версиями.

9.9.2. Обход кода

Большинство реально используемых макросов принимают выражение, делают немного замен с перестановками в нем и возвращают результат. Этим весьма простым макросам не требуется выполнять глубокий анализ своих аргументов. Но вот макрос, к примеру, преобразующий арифметические выражения из инфиксной записи в привычную для Лиспа префиксную, однозначно потребует более сложной логики. Рассмотрим макрос `with-slots` из CLOS на примере Meroonet. К полям объекта — для определённости, скажем, к полям объекта класса `Point` — доступ обеспечивают функции-аксессоры вроде `Point-x`. Гораздо удобнее было бы обращаться к полям просто по имени: `x` или `y`; по крайней мере, в контексте объявления методов, где нужный объект самоочевиден. Тогда можно будет объявлять методы как в Smalltalk [GR83]:

```
(define-handy-method (double (o Point))
  (set! x (* 2 x))
  (set! y (* 2 y))
  o )
```

вместо такого ужаса:

```
(define-method (double (o Point))
  (set-Point-x! o (* 2 (Point-x o)))
  (set-Point-y! o (* 2 (Point-y o)))
  o )
```

Макрос `define-handy-method`, соответственно, должен пройтись по своему телу и заменить все ссылки и присваивания переменным, чьи имена совпадают с именами полей класса, на корректные обращения к данным полям. Он мог бы воспользоваться более быстрыми вариантами аксессоров, которые не выполняют всяческих проверок типов, так как правильный тип аргумента уже гарантирован дискриминантом метода. Подобный макрос не только улучшает читабельность, но и немного помогает компилятору с оптимизациями.

Однако перед тем, как раскатывать губу, сначала стоит вспомнить, что тело метода, передаваемое в `define-handy-method`, — это не программа, а просто S-выражение, в котором могут быть и другие макросы. Очевидно, что сначала надо раскрыть их, а для этого потребуется доступ к макроэкспандеру. Более того, как вы помните, раскрытие макросов может вызвать создание других макросов, существуют локальные макросы и так далее, так что это должен быть именно текущий макроэкспандер со всем своим состоянием. Дабы не уходить в дебри реализации, предположим, что во время раскрытия макроса мы всегда имеем доступ к его экспандеру посредством переменной `macroexpand`.

После раскрытия макросов тело `define-handy-method` уже является нормальной программой, которую можно анализировать. Это несложно. В конце концов, всю эту книгу мы только тем и занимались, что анализировали программы. Сейчас нас интересуют специальные формы и вызовы функций, где могут встретиться необходимые переменные. Цитаты можно не трогать, там переменных нет. Смотреть надо на локальные связывающие формы, способные ввести новые переменные, которые скроют наши удобные ссылки на поля объекта; очевидно, такие переменные тоже не надо трогать. Реализовать собственно обход программ на Лиспе несложно, см. [Cur89, Wat93]; значительно сложнее будет определить, на какие формы языка — вернее, его реализации — следует реагировать, а на какие нет.

Ядром языка являются специальные формы, но в их толковании многие реализации часто допускают определённые вольности, как то:

- стандартные возможности, реализованные в виде специальных форм (чаще всего `let` и `letrec`);
- введение нестандартных специальных форм (`define-class`, например);
- реализация некоторых специальных форм как макросов (вроде известной вам `begin`).

Не имея рефлексивной информации о языке, сложно ответить на возникающие в процессе обхода вопросы: «Во что именно превратились все формы `begin`? Как найти в исходном коде условные выражения, если `if` — это на самом деле хитрый макрос, раскрывающийся в специальную форму `typecase`? Что делать с незнакомыми специальными формами вроде `define-class`, чей синтаксис объявления полей идентичен синтаксису вызова функций? Откуда

экспандер узнает, что `bind-exit` является связывающей формой, которая создаёт локальные привязки, способные скрывать одноимённые переменные?»

Хорошо было бы заморозить набор специальных форм и явно запретить реализациям изменять его в большую или меньшую сторону. В таком случае можно быть уверенным, что после окончания раскрытия макросов программа имеет чётко определённую структуру, где не осталось всяких специальных встроенных макросов или чего-то подобного. К сожалению, в Scheme нет стандартного способа получить необходимую рефлексивную информацию о языке и его реализации, так что написать идеально переносимую версию `define-handy-method` не получится.

9.10. Непредвиденные захваты

Идея *гигиены* применительно к макросам была тщательно рассмотрена в работах [KFFD86, BR88, CR91a]. Проблема, решаемая ею, состоит в том, что макросы после раскрытия могут содержать переменные, которые в некотором смысле «свободны», то есть ни с чем не связаны, а значит, являются источником трудноуловимых ошибок, возникающих из-за сокрытия как переменных окружающего кода свободными переменными макросов, так и наоборот. Следующий пример демонстрирует оба случая:

```
(define-abbreviation (acons key value alist)
  `(let ((f cons)) (f (f ,key ,value) ,alist)) )

(let ((cons list)
      (f #f) )
  (acons 'false f '()) )
```

Переменная `cons`, оказавшись в раскрытом макросе `acons`, будет ссылаться не на ту `cons`, к которой все привыкли, а на локальную переменную `cons`, введённую формой `let`, внутри которой расположен макровызов `acons`. И наоборот, сам макрос `acons` вводит переменную `f`, которая помешает аргументу `value` получить ожидаемое значение `#f`. Ужас!

Гигиеничные макросы в принципе лишены подобных проблем. Давайте подумаем, как им это удаётся.

Вот уже тридцать лет от второй проблемы в Лиспе избавляются с помощью простого переименования переменных. Раз переменная `f` внутри макроса может пересечься с какой-то внешней лексической переменной, то сделаем так, чтобы этого никогда не произошло: возьмём `gensym` и получим для внутренней переменной неповторимое и гарантированно уникальное в текущем лексическом окружении имя. Поражаясь простоте решения, невозмутимо пишем:

```
(define-abbreviation (acons key value alist)
  (let ((f (gensym)))
    `(let ((,f cons)) (,f (,f ,key ,value) ,alist)) ) )
```

С первой проблемой, касающейся ссылки на `cons` в макросе, разобраться будет сложнее. В данном случае хочется иметь какой-то механизм, с помощью которого можно было бы сказать, что `cons` — это ссылка на глобальную переменную `cons`, а не на что-то другое. В конце концов, именно эта переменная `cons` была видна с места определения макроса `acons`. Из этого наблюдения следует главное правило макрогигиены: в раскрытом макросе свободные переменные сохраняют смысл, который они имели при определении этого макроса.

На случай с `cons` во многих реализациях Лиспа и Scheme есть механизм доступа к глобальным переменным в любом контексте; обычно это нечто вроде `(global cons)` или `lisp:cons`. Но иногда необходимо связать локальную переменную с переменной в раскрытом макросе, как в следующем примере:

```
(let ((results '())
      (compose cons) )
  (let-syntax ((push (syntax-rules ()
                       ((push e) (set! results (compose e results))) )))
    π
    results ) )
```

Мы хотим, чтобы во всём теле π этого выражения можно было использовать макрос `push`, который присоединяет свой аргумент к переменной `results` с помощью функции `compose`. Для этого необходимо обеспечить гигиеничность данного макроса — сделать так, чтобы ни одно определение внутри π не смогло изменить смысл `push`. По существу, у нас есть два варианта:

1) переименовать все мешающие переменные внутри π так, чтобы всегда были видны правильные `results` и `compose`;

2) аккуратно переименовать сами переменные `results` и `compose`, чтобы они не пересекались ни с чем из π; то есть, изменить `let`-форму, определение `push` и тело `let-syntax`.

Несмотря на то, что мы постоянно говорим о переменных и привязках, для макросов таких вещей не существует. Они не захватывают привязки подобно замыканиям, так как никаких привязок вообще ещё нет — ведь во время раскрытия макросов всё это просто S-выражения! Более того, R^5RS не вводит никаких зарезервированных идентификаторов, поэтому ничто не запрещает создавать макросы-тёзки специальных форм. Следовательно, внутренние макросы π вполне могут «переопределить» примитив `set!`, очевидно свободный в макросе `push`. Даже это не должно изменить поведения `push`. Гигиеничные макросы захватывают сам *смысл* символов, а не значения переменных.

Гигиена — это замечательная и привлекательная возможность, но её нельзя принять с распростёртыми объятиями всюду, так как существуют чрезвычайно полезные, но не полностью гигиеничные макросы. Наиболее известный из них — это макрос `loop`. Из него обычно можно выйти с помощью функции `exit`. Однако, если он будет реализован вот так:

```
(define-syntax loop
  (syntax-rules ()
    ((loop e1 e2 ...)
     (call/cc (lambda (exit)
                (let loop () e1 e2 ... (loop)) )) ) ) )
```

то из него нельзя будет выйти таким образом, потому что любое упоминание `exit` внутри форм `e1`, `e2`... из-за гигиенических переименований будет ссылаться не на тот `exit`, который объявлен внутри макроса, а на тот, который виден с места макровызова. Здесь необходимо как-то сказать экспандеру, чтобы он не трогал переменную `exit`, но в стандарте R^5RS такая возможность не предусмотрена. Строгая гигиена является хорошим решением по умолчанию: в определении `loop` кажется логичным, что форма `(loop)` ссылается именно на определяемую там же локальную переменную; однако иногда от правил требуется отступать.

В Интернете можно найти множество вполне неплохих реализаций гигиеничных макросистем, каждая из них обладает своими особенностями, достойными изучения. Приводимое здесь решение проблемы гигиены не ограничивает вычисления, допускаемые во время раскрытия макросов, но является довольно низкоуровневым по сравнению с другими известными реализациями, например, описанными в [KCR98, DHB93]. Идея заключается в явном перечислении символов, чей смысл во время раскрытия должен остаться тем же, каким он был во время объявления макроса, а не определяться из контекста. Рассмотрим на примере. Форма `with-aliases` принимает список пар «имя переменной — символ»; *на время раскрытия* макросов эта форма связывает переменные с тем, что означают соответствующие символы в текущем окружении. Внутри макросов можно пользоваться данными переменными, чтобы получить доступ к необходимым объектам. Остальные аргументы формы `with-aliases` составляют её тело, лексический блок, содержащий созданные переменные. Итого, предыдущий пример переписывается следующим образом:

```
(let ((results '())
      (compose cons) )
  (with-aliases ((s set!) (r results) (c compose))
    (let-abbreviation (((push e)
                        `(,s ,r (,c ,e ,r)) ))
      π
      results ) ) )
```

А пример с `loop` теперь выглядит вот так:

```
(with-aliases ((cc call/cc) (lam lambda) (ll let))
  (define-abbreviation (loop . body)
    (let ((loop (gensym)))
      `(,cc (,lam (exit)
                  (,ll ,loop () ,@body (,loop)) )) ) ) )
```

Все символы, чей смысл мы хотим сохранить неизменным при раскрытии макросов, перечисляются в `with-aliases`. Она захватывает их, а во время раскрытия макросов мы можем воспользоваться новыми именами, чтобы вставить на их место именно то, что означали эти имена при входе в форму `with-aliases`, а не то, что они означают в текущем окружении, где раскрывается макрос.

Квазицитирование при такой реализации не особо удобно ввиду того, что большая часть выражения не фиксирована. Также этот механизм довольно низкоуровневый — он не делает макросы автоматически гигиеничными, лишь предоставляет инструмент для самостоятельного соблюдения гигиены. Следующий раздел посвящён подробному разбору этой и других частей нашей макросистемы.

9.11. Макросистема

В этом разделе мы рассмотрим реализацию макросистемы, которая предоставляет следующие возможности:

- `define-abbreviation` для определения глобальных макросов;
- `let-abbreviation` для определения локальных макросов;
- `eval-in-abbreviation-world` для вычислений в макромире;
- `with-aliases` для захвата смысла символов.

Чтобы ещё раз подчеркнуть различие между подготовкой и исполнением программ, результат раскрытия макросов будет на лету преобразовываться в дерево объектов. Получаемое промежуточное объектное представление программ можно использовать по-разному: можно отдать его на непосредственное исполнение быстрому интерпретатору, как в шестой главе [**см. стр. 222**], а можно можно, подобно компилятору из десятой главы [**см. стр. 427**], преобразовать дерево объектов в код на Си. В любом случае макросы раскрываются лишь один раз, после чего структура программы застывает в виде объектов.

9.11.1. Объектификация

Отчасти объектификация схожа с реификацией — оба процесса приводят к некоему «осязаемому» результату. Под объектификацией понимается преобразование программ в соответствующие им объекты. Также сюда входит проверка и нормализация синтаксиса обрабатываемых выражений, дабы нас в дальнейшем не тревожили глупые ошибки. Эта трансформация программ схожа с той, что выполняется для быстрого интерпретатора (и намерения

у них тоже схожи), но в этот раз она выполняется исключительно ради получения промежуточного представления как такового. Поэтому вместо замыканий без аргументов, которые удобно вызывать, но очень сложно анализировать, мы возьмём нечто более прозрачное — объекты, чьи поля будут содержать всю необходимую для вычислений информацию. Кроме того, объектами гораздо проще манипулировать, используя обобщённые функции.

Итак, иерархия используемых классов:

```
(define-class Program                (Object ())
(define-class Reference              (Program (variable))
(define-class Local-Reference        (Reference ())
(define-class Global-Reference       (Reference ())
(define-class Predefined-Reference   (Reference ())
(define-class Global-Assignment      (Program (variable form))
(define-class Local-Assignment       (Program (reference form))
(define-class Function               (Program (variables body))
(define-class Alternative Program (condition consequent alternant))
(define-class Sequence               (Program (first last))
(define-class Constant               (Program (value))
(define-class Application            (Program ())
(define-class Regular-Application    (Application (function arguments))
(define-class Predefined-Application (Application (variable arguments))
(define-class Fix-Let                (Program (variables arguments body))
(define-class Arguments              (Program (first others))
(define-class No-Argument            (Program ())
(define-class Variable               (Object (name))
(define-class Global-Variable        (Variable ())
(define-class Predefined-Variable    (Variable (description))
(define-class Local-Variable         (Variable (mutable? dotted?))
```

В этом перечне легко заметить влияние предыдущих глав на наши представления о внутреннем устройстве языка. Например, вызовы примитивов, приводимых форм [**см. стр. 237**] и обычных функций выделены в отдельные категории. Класс `Program` представляет исключительно вычислимые понятия: обращения к переменным, присваивания, вызовы функций. Сами переменные — привязки значений к именам — являются объектами класса `Variable`, это не программы. Также стоит обратить внимание на деталь, отличающую эту реализацию от рассмотренных ранее. Большинство определённых классов содержат столько полей, сколько синтаксических элементов имеют соответствующие формы языка. Однако класс локальных переменных содержит два дополнительных булевых поля: первое показывает, обнаружены ли в программе присваивания этой переменной, а второе истинно, если данная переменная — это точечный аргумент. Как вы помните, точечные переменные требуют особого внимания при обработке вызовов функций.

Часто предобработчик считывает исходные выражения просто с помощью `read` (или эквивалентной функции, возвращающей сырые S-выражения). Од-

нако более удобным подходом было бы использование функции, формирующей четвёрку из собственно выражения, файла, откуда оно было считано, а также соответствующих строки и столбца. Тогда мы сможем выводить прекрасно детализированные сообщения об ошибках. Более того, в символы можно добавить дополнительное поле со ссылкой на экспандер одноимённого макроса, что ускорит поиск макросов. На этапе предобработки допустимо расширять поля объектов как угодно, если это поможет улучшить её скорость и/или качество.

Итак, считанные S-выражения анализируются и преобразуются в объект класса `Program`, при этом в нём раскрываются все встреченные макросы. Главная функция `objectify` принимает два аргумента: S-выражение и его лексическое окружение. Если выражение является формой, то `objectify` смотрит на её функциональный терм и выполняет соответствующую ему обработку. Обработчики макросов не разбросаны где попало по коду, а располагаются в поле `handler` объектов класса `Magic-Keyword` (терминология [SS75]), туда же для удобства отправлены и специальные формы.

```
(define-class Magic-Keyword Object (name handler))

(define (objectify e r)
  (if (atom? e)
      (cond ((Magic-Keyword? e) e)
            ((Program? e)       e)
            ((symbol? e)        (objectify-symbol e r))
            (else               (objectify-quotation e r)) )
      (let ((m (objectify (car e) r)))
        (if (Magic-Keyword? m)
            ((Magic-Keyword-handler m) e r)
            (objectify-application m (cdr e) r) ) ) ) )
```

Теперь остаётся лишь разобрать всевозможные специализированные подфункции, выполняющие преобразования. Большинство из них просто заполняют поля объектов результатами рекурсивной объектификации элементов соответствующих S-выражений. И правда, сложно придумать какой-то другой способ для `objectify-alternative` и `objectify-sequence`. Помимо этого функция `objectify-sequence` нормализует последовательности форм, превращая их в набор одинаковых по структуре бинарных последовательностей, подобных `cons`-парам.

```
(define (objectify-quotation value r)
  (make-Constant value) )

(define (objectify-alternative ec et ef r)
  (make-Alternative (objectify ec r)
                    (objectify et r)
                    (objectify ef r) ) )
```

```
(define (objectify-sequence e* r)
  (if (pair? e*)
      (if (pair? (cdr e*))
          (let ((a (objectify (car e*) r)))
            (make-Sequence a (objectify-sequence (cdr e*) r)) )
          (objectify (car e*) r) )
      (make-Constant 42) ) )
```

Обработка вызовов функций немного сложнее, так как необходимо проанализировать выражение и категоризовать его как приводимую форму, вызов примитива или вызов обычной функции. Анализ выполняется на основе первого терма формы. Единственная возможная тут неясность была упомянута ранее [**см. стр. 239**]: доступна ли нам информация об арности предопределённых функций и как её использовать для более эффективной компиляции. Класс `Functional-Description` подробно рассматривается чуть позже (в разделе 10.8.6) вместе с механизмом наполнения начального окружения.

```
(define (objectify-application ff e* r)
  (let ((ee* (convert2arguments (map (lambda (e)
                                       (objectify e r) ) e*))))
    (cond ((Function? ff)
           (process-closed-application ff ee*) )
          ((Predefined-Reference? ff)
           (let* ((fvf  (Predefined-Reference-variable ff))
                  (desc (Predefined-Variable-description fvf)) )
             (if (Functional-Description? desc)
                 (if ((Functional-Description-comparator desc)
                      (length e*) (Functional-Description-arity desc) )
                     (make-Predefined-Application fvf ee*)
                     (objectify-error
                      "Incorrect predefined arity" ff e* ) )
                 (make-Regular-Application ff ee*) ) ) )
          (else (make-Regular-Application ff ee*)) ) ) )

(define (process-closed-application f e*)
  (let ((v* (Function-variables f))
        (b  (Function-body f)) )
    (if (and (pair? v*) (Local-Variable-dotted? (car (last-pair v*))))
        (process-nary-closed-application f e*)
        (if (= (number-of e*) (length v*))
            (make-Fix-Let v* e* b)
            (objectify-error "Incorrect regular arity" f e*) ) ) ) )
```

Список аргументов вызова функции тоже преобразуется в объект (экземпляр класса `Arguments`); заканчивается этот список на специальный объект класса `No-Argument`. Количество аргументов можно определить с помощью обобщённой функции `number-of`.

```
(define (convert2arguments e*)
  (if (pair? e*)
      (make-Arguments (car e*) (convert2arguments (cdr e*)))
      (make-No-Argument) ) )

(define-generic (number-of (o)))

(define-method (number-of (o Arguments))
  (+ 1 (number-of (Arguments-others o))) )

(define-method (number-of (o No-Argument)) 0)
```

Как обычно, редкий и громоздкий случай n-арных приводимых форм обрабатывается отдельно. Дополнительные аргументы собираются в список, а соответствующая точечная переменная заменяется обычной. [**см. стр. 237**]

```
(define (process-nary-closed-application f e*)
  (let* ((v* (Function-variables f))
         (b  (Function-body f))
         (o (make-Fix-Let
             v*
             (let gather ((e* e*) (v* v*))
               (if (Local-Variable-dotted? (car v*))
                   (make-Arguments
                    (let pack ((e* e*))
                      (if (Arguments? e*)
                          (make-Predefined-Application
                           (find-variable? 'cons g.predef)
                           (make-Arguments
                            (Arguments-first e*)
                            (make-Arguments
                             (pack (Arguments-others e*))
                             (make-No-Argument) ) ) )
                          (make-Constant '()) ) )
                    (make-No-Argument) )
                   (if (Arguments? e*)
                       (make-Arguments (Arguments-first e*)
                                       (gather (Arguments-others e*)
                                               (cdr v*) ) )
                       (objectify-error
                        "Incorrect dotted arity" f e* ) ) ) )
             b )) )
    (set-Local-Variable-dotted?! (car (last-pair v*)) #f)
    o ) )
```

Абстракции анализируются и объектифицируются функцией `objectify-function`. Она использует `objectify-variables-list` для обработки списка аргументов, которыми расширяется окружение создаваемого замыкания.

```
(define (objectify-function names body r)
  (let* ((vars (objectify-variables-list names))
         (b    (objectify-sequence body (r-extend* r vars))) )
    (make-Function vars b) ) )

(define (objectify-variables-list names)
  (if (pair? names)
      (cons (make-Local-Variable (car names) #f #f)
            (objectify-variables-list (cdr names)) )
      (if (symbol? names)
          (list (make-Local-Variable names #f #t))
          '() ) ) )
```

Наконец, `objectify-symbol` отвечает за обработку переменных. Сначала она пытается найти их в текущем едином статическом лексическом окружении — `r`. Если переменная там не обнаружена, то она добавляется в глобальное окружение функцией `objectify-free-global-reference`. Верно, здесь мы решили создавать новые переменные автоматически при их первом упоминании.

```
(define (objectify-symbol variable r)
  (let ((v (find-variable? variable r)))
    (cond ((Magic-Keyword? v)       v)
          ((Local-Variable? v)      (make-Local-Reference v))
          ((Global-Variable? v)     (make-Global-Reference v))
          ((Predefined-Variable? v) (make-Predefined-Reference v))
          (else (objectify-free-global-reference variable r)) ) ) )

(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name)))
    (insert-global! v r)
    (make-Global-Reference v) ) )
```

Окружение `r` является, в принципе, списком локальных переменных, за которым следует список всех глобальных переменных, а в конце располагается список предопределённых переменных. Так как это статическое окружение, то в нём отсутствуют значения переменных — они определяются динамически, во время исполнения программы. Различные окружения представляются списками объектов класса `Environment`. Их можно расширять как локальными, так и глобальными переменными. Новые глобальные переменные физически добавляются в общую глобальную часть, до которой можно добраться с помощью функции `find-global-environment`.

```
(define-class Environment      Object (next))
(define-class Full-Environment Environment (variable))

(define (r-extend* r vars)
  (if (pair? vars) (r-extend (r-extend* r (cdr vars)) (car vars))
                   r ) )
```

```
(define (r-extend r var)
  (make-Full-Environment r var) )

(define (find-variable? name r)
  (if (Full-Environment? r)
      (let ((var (Full-Environment-variable r)))
        (if (eq? name (cond ((Variable? var)
                             (Variable-name var))
                            ((Magic-Keyword? var)
                             (Magic-Keyword-name var) ) ) )
            var
            (find-variable? name (Full-Environment-next r)) ) )
      (if (Environment? r)
          (find-variable? name (Environment-next r))
          #f ) ) )

(define (insert-global! variable r)
  (let ((r (find-global-environment r)))
    (set-Environment-next!
     r (make-Full-Environment (Environment-next r) variable) ) ) )

(define (mark-global-preparation-environment g)
  (make-Environment g) )

(define (find-global-environment r)
  (if (Full-Environment? r)
      (find-global-environment (Full-Environment-next r))
      r ) )
```

Обработка присваиваний позволяет собрать информацию об изменяемости локальных переменных: именно здесь у них устанавливается флаг `mutable?`. Эта информация впоследствии будет использована для проведения оптимизаций в десятой главе. [**см. стр. 427**]

```
(define (objectify-assignment variable e r)
  (let ((ov (objectify variable r))
        (of (objectify e r)) )
    (cond ((Local-Reference? ov)
           (set-Local-Variable-mutable?!
            (Local-Reference-variable ov) #t )
           (make-Local-Assignment ov of) )
          ((Global-Reference? ov)
           (make-Global-Assignment (Global-Reference-variable ov) of) )
          (else
           (objectify-error "Illegal assignment" variable) ) ) ) )
```

Для программ, представленных деревьями подобных объектов, очень легко написать интерпретатор, проверяющий правильность выполнения преобразования. Результат будет сильно напоминать объектный интерпретатор из третьей главы и быстрый интерпретатором из шестой.

Рассмотрим общую структуру этого интерпретатора, чтобы вы не терялись в последующих фрагментах кода. Вычисления выполняет функция `evaluate`. Она принимает два аргумента: экземпляр `Program` и экземпляр `Environment` (теперь уже A-список пар из имён переменных и значений). Вначале окружение содержит только предопределённые переменные. Его статическая часть представлена переменной `g.predef`, а динамическая — `sg.predef`. Динамическая часть связывает экземпляры `RunTime-Primitive` с предопределёнными функциями. Расширяется динамическое окружение с помощью функции `sr-extend`.

9.11.2. Специальные формы

В данный момент объектификатор не распознаёт специальные формы, так что для каждой из них надо определить персональную процедуру преобразования в соответствующий объект класса `Program`. Они определяются как ключевые слова со специальными обработчиками. Эти обработчики будут просто вызывать функции, рассмотренные в предыдущем разделе.

```
(define special-if
  (make-Magic-Keyword 'if
    (lambda (e r)
      (objectify-alternative (cadr e) (caddr e) (cadddr e) r) ) ) )

(define special-begin
  (make-Magic-Keyword 'begin
    (lambda (e r)
      (objectify-sequence (cdr e) r) ) ) )

(define special-quote
  (make-Magic-Keyword 'quote
    (lambda (e r)
      (objectify-quotation (cadr e) r) ) ) )

(define special-set!
  (make-Magic-Keyword 'set!
    (lambda (e r)
      (objectify-assignment (cadr e) (caddr e) r) ) ) )

(define special-lambda
  (make-Magic-Keyword 'lambda
    (lambda (e r)
      (objectify-function (cadr e) (cddr e) r) ) ) )
```

Естественно, точно таким же образом можно определить столько дополнительных специальных форм, сколько необходимо. Чтобы ничего потом не напутать, запишем их все в список `*special-form-keywords*`:

```
(define *special-form-keywords*
  (list special-quote
        special-if
        special-begin
        special-set!
        special-lambda
        ; ; cond, letrec и т. д.
        special-let
        ) )
```

9.11.3. Уровни интерпретации

Как вы знаете, эндогенное раскрытие макросов подразумевает наличие внутри специального вычислителя. Но не забывайте, что эндогенный подход не требует единства мира: результат объектификации не обязательно исполняется в том же окружении, где он создавался. Компилятор в Си из десятой главы [**см. стр. 427**] служит этому хорошим примером. Нам бы не хотелось запутаться во всех этих вычислителях, так что давайте введём понятие *уровня интерпретации* (или вычислений) и примем наиболее строгий постулат: язык, макроязык, макроязык макроязыка и так далее — все они отделены друг от друга, соответствующие вычисления производятся в различных глобальных окружениях. Данные вычислители представляются объектами класса `Evaluator`, описывающего их важнейшие свойства:

```
(define-class Evaluator Object
  ( mother
    Preparation-Environment
    RunTime-Environment
    eval
    expand
    ) )
```

Взаимодействие вычислителей весьма запутанно. Рисунок 9.9 иллюстрирует цепочку вызовов, возникающую в процессе раскрытия макросов. Экспандер одного уровня использует вычислитель следующего, который в свою очередь тоже предварительно раскрывает макросы в получаемом выражении, используя вычислитель следующего уровня, и так далее. Цепочка прерывается, как только получаемое выражение оказывается лишённым макросов; в таком случае следующий уровень не нужен. Обычно бывает достаточно всего двух-трёх уровней. Каждый уровень интерпретации имеет своё окружение времени подготовки (для `expand`) и окружение времени исполнения (для `eval`), кроме разве что самого нижнего уровня, где в цепочке может участвовать один лишь экспандер.

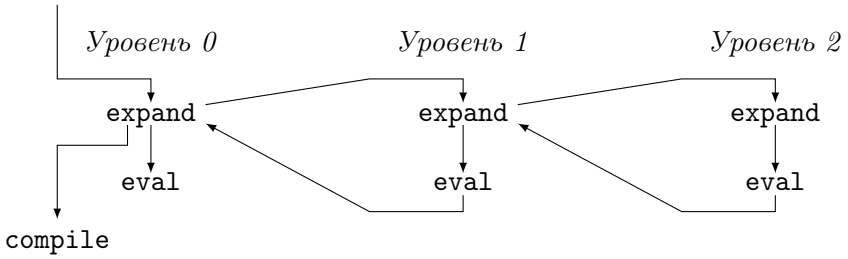


Рис. 9.9. Цепочка вычислителей.

Функция `create-evaluator` используется для создания новых уровней интерпретации. Она строит новый уровень над тем, который получает через аргумент. Также она определяет пару функций `eval` и `expand`, а также соответствующие им окружения подготовки и исполнения. Перед началом вычислений `eval` раскрывает в полученном выражении макросы, затем выполняет собственно вычисления с помощью функции `evaluate`. Этому чистому вычислителю кроме выражения необходимо лишь окружение времени исполнения, хранимое в поле `RunTime-Environment` текущего экземпляра `Evaluator`. При необходимости вычислитель может физически изменять данное окружение. Функция `expand` объектифицирует получаемое выражение с помощью `objectify` в окружении времени подготовки, хранимом в соответствующем поле `Preparation-Environment`. Для простоты все глобальные переменные, создаваемые во время вложенных вычислений, помещаются в персональное глобальное окружение текущего вычислителя с помощью функции `enrich-with-new-global-variables!`. Функция `eval` рефлексивным образом устанавливается в своё собственное окружение времени исполнения как предопределённый примитив. Таким образом, различные уровни пользуются различными функциями `eval`. Далее, перед началом работы окружение времени подготовки расширяется специальными формами, хранимыми в переменной `*special-form-keywords*`. Эта переменная должна быть видима на всех уровнях. Кроме того, окружение времени подготовки расширяется предопределёнными макросами, возвращаемыми вызовом `(make-macro-environment level)`, который мы рассмотрим чуть позже.

```
(define (create-evaluator old-level)
  (let ((level 'wait)
        (g     g.predef)
        (sg    sg.predef))
    (define (expand e)
      (let ((prg (objectify
                  e (Evaluator-Preparation-Environment level) )))
        (enrich-with-new-global-variables! level)
        prg ) )
    (define (eval e)
      (let ((prg (expand e)))
        (evaluate prg (Evaluator-RunTime-Environment level)) ) )
```

```
;; Создаём вычислитель
  (set! level (make-Evaluator old-level 'wait 'wait eval expand))

;; Наполняем глобальное окружение
  (set! g (r-extend* g *special-form-keywords*))
  (set! g (r-extend* g (make-macro-environment level)))
  (let ((eval-var (make-Predefined-Variable
                   'eval (make-Functional-Description = 1 "") ))
        (eval-fn (make-RunTime-Primitive eval = 1)) )
    (set! g (r-extend g eval-var))
    (set! sg (sr-extend sg eval-var eval-fn)) )

;; Устанавливаем окружения в вычислитель
  (set-Evaluator-Preparation-Environment!
   level (mark-global-preparation-environment g) )
  (set-Evaluator-RunTime-Environment!
   level (mark-global-runtime-environment sg) )

  level ) )
```

Если программе потребуется явный экспандер или вычислитель, то достаточно создать новый уровень интерпретации с помощью `create-evaluator` и достать оттуда необходимую функцию `expand` или `eval`. Осторожно: вопреки ожиданиям, `expand` возвращает экземпляры `Program`, а не выражения Scheme. Если нужны именно такие выражения, то несложно написать соответствующую функцию-преобразователь. [**см. упр. 9.4**]

9.11.4. Макросы

По нашей задумке, в начальном окружении времени подготовки должны изначально присутствовать предопределённые макросы, создаваемые функцией `make-macro-environment`. Вот эта четвёрка: `eval-in-abbreviation-world`, `define-abbreviation`, `let-abbreviation` и `with-aliases`. Все они предполагают существование следующего уровня интерпретации, который должен быть построен той же функцией `make-macro-environment`. Однако, если попытаться создать этот уровень сразу же, то мы погрязнем в бесконечном цикле, так как на новом уровне тоже должна быть своя четвёрка предопределённых макросов. Эта проблема известна в философии как *бесконечная регрессия*. Новый уровень интерпретации необходим только в том случае, если какой-то из данных макросов действительно используется на текущем уровне. [**см. упр. 9.3**] Поэтому мы просто пообещаем создать этот уровень, когда он кому-то понадобится. Если это так, то новый уровень останется и продолжит накапливать в себе определения, иначе же он просто никогда не будет создан.

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay (create-evaluator current-level))))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel) )
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel) )
          (make-Magic-Keyword 'with-aliases
           (special-with-aliases metalevel) ) ) ) )
```

Макрос `eval-in-abbreviation-world` является наиболее простым, так как его работа сводится лишь к вычислению выражения с помощью вычислителя следующего уровня. Вот здесь как раз и придётся потребовать создать для нас данный уровень. Возвращаемым значением `eval-in-abbreviation-world` должен быть объект, получаемый вызовом объектификатора в текущем окружении. Это правило общее для всех обитателей мира макросов.

```
(define (special-eval-in-abbreviation-world level)
  (lambda (e r)
    (let ((body (cdr e)))
      (objectify ((Evaluator-eval (force level))
                  `(,special-begin . ,body) ) r) ) ) )
```

Макрос `define-abbreviation` создаёт и переопределяет глобальные макросы. Он конструирует соответствующий экспандер и вычисляет полученную абстракцию на следующем уровне, затем на текущем уровне в глобальном окружении предобработки создаётся новое ключевое слово, чьим обработчиком назначается полученный ранее экспандер. Функция `invoke` в данном случае более удобна для вычислений, нежели `evaluate`. Когда создаваемый макрос вызовет экспандер с помощью `invoke`, он будет проводить вычисления в том окружении времени исполнения, где был создан, то есть в окружении следующего уровня, которое замкнуто в экспандере. Возвращает же макрос `define-abbreviation` просто `#t`. Можно было бы возвращать символ с именем созданного макроса, но мы сэкономим немного памяти на этом.

```
(define (special-define-abbreviation level)
  (lambda (e r)
    (let* ((call      (cadr e))
           (body      (cddr e))
           (name      (car call))
           (variables (cdr call)) )
      (let ((expander ((Evaluator-eval (force level))
                       `(,special-lambda ,variables . ,body) )))
        (define (handler e r)
          (objectify (invoke expander (cdr e)) r) )
        (insert-global! (make-Magic-Keyword name handler) r)
        (objectify #t r) ) ) ) )
```

Локальные макросы создаются похожим образом. Разница только в том, что новые ключевые слова помещаются в начало текущего окружения — туда, где расположены локальные определения.

```
(define (special-let-abbreviation level)
  (lambda (e r)
    (let ((level  (force level))
          (macros (cadr e))
          (body   (cddr e)) )
      (define (make-macro def)
        (let* ((call      (car def))
               (body      (cdr def))
               (name      (car call))
               (variables (cdr call)) )
          (let ((expander ((Evaluator-eval level)
                           `(,special-lambda ,variables . ,body) )))
            (define (handler e r)
              (objectify (invoke expander (cdr e)) r) )
            (make-Magic-Keyword name handler) ) ) )
      (objectify `(,special-begin . ,body)
                 (r-extend* r (map make-macro macros)) ) ) ) )
```

Самым сложным из всей четвёрки является макрос `with-aliases`, так как его работа затрагивает сразу несколько уровней. Он должен захватить смысл символов на текущем уровне, передать его на следующий, там связать эти значения с новыми переменными, причём сделать всё это исключительно на время выполнения раскрытия своего тела. Именно хореография областей видимости, уровней интерпретации и времени жизни делает этот макрос столь сложным.

```
(define (special-with-aliases level)
  (lambda (e current-r)
    (let* ((level   (force level))
           (old-r   (Evaluator-Preparation-Environment level))
           (old-sr  (Evaluator-RunTime-Environment level))
           (aliases (cadr e))
           (body    (cddr e)) )
      (let bind ((aliases aliases)
                 (r  old-r)
                 (sr old-sr) )
        (if (pair? aliases)
            (let* ((variable (car (car aliases)))
                   (keyword  (cadr (car aliases)))
                   (var      (make-Local-Variable variable #f #f)) )
              (bind (cdr aliases)
                    (r-extend r var)
                    (sr-extend sr var
                               (objectify keyword current-r) ) ) )
```

```
(let ((result 'wait))
  (set-Evaluator-Preparation-Environment! level r)
  (set-Evaluator-RunTime-Environment! level sr)
  (set! result (objectify `(,special-begin . ,body)
                          current-r ))
  (set-Evaluator-Preparation-Environment! level old-r)
  (set-Evaluator-RunTime-Environment! level old-sr)
  result ) ) ) ) ) )
```

Для временной модификации окружения следующего уровня интерпретации прекрасно подошёл бы механизм динамического связывания, гарантирующий восстановление старого окружения после завершения раскрытия формы `with-aliases`.

Переменные, создаваемые `with-aliases`, связываются со значениями, которые ранее вернула функция `objectify`, то есть с объектами класса `Program` или `Magic-Keyword`. Так как подобные объекты вполне могут быть возвращены обратно из `with-aliases` в результате раскрытия макросов, то `objectify` должна уметь с ними обращаться. Именно поэтому она вначале проверяет, не является ли полученное выражение уже объектифицированной программой или ключевым словом. [**см. стр. 412**]

9.11.5. Ограничения

Механизм гигиеничного переименования позволяет жонглировать переменными и привязками, даёт возможность захватывать значение символа в одном месте, а использовать его в совершенно другом — там, откуда до данной переменной нельзя добраться иным способом; и даже там, где она вовсе не существует. Например:

```
(let ((count 0))
  (with-aliases ((c count))
    (define-abbreviation (tick) c) )
  (tick) )
(let ((count 1) (c 2))
  (tick) )
```

Глобальный макрос `tick` ссылается на локальную переменную `count`, которая не видна из второй формы `let`. Действительно, необдуманный захват смысла символов может вызывать ошибки нового типа: обращения к несуществующим переменным. Несмотря на то, что в локальном окружении есть переменная с именем `count`, это *не та* `count`, которую ищет `tick`! Причина ошибки видна яснее в следующем «деобъектифицированном» представлении исходного фрагмента:

```
((LAMBDA (COUNT501)
   (BEGIN #T ;; (tick) ⇝ COUNT501
          COUNT501 ) ) 0)
((LAMBDA (COUNT502 C503) COUNT501) 1 2)
```

Расположение формы `with-aliases` необходимо тщательно выбирать, так как это нечто вроде `let`, только для следующего уровня интерпретации. Если написать

```
(define-abbreviation (loop . body)
  (with-aliases ((cc call/cc) (lam lambda) (ll let))
    (let ((loop (gensym)))
      `(,cc (,lam (exit) (,ll ,loop () ,@body (,loop)))) ) ) )
```

то форма `with-aliases` захватит смысл символов на уровне макроопределений, так что использовать их можно только на этом или более высоком уровне. Здесь `with-aliases` содержится внутри `define-abbreviation`, поэтому в момент раскрытия макроса `loop` смысл `call/cc` захватывается в мире макросов, привязывается к переменной `cc` и… мы получаем ошибку `"cc: unknown value"`, так `call/cc` из мира макросов недоступна раскрытому коду, находящемуся в мире программ.

Если не брать в расчёт специальные формы и другие ключевые слова вроде `else` и `=>`, то смысл локальных и глобальных переменных вполне успешно захватывался с помощью полноценных окружений и форм `import`/`export`. [**см. стр. 354**] Однако механизм, показанный в данной главе, несомненно является более мощным, так как, во-первых, он позволяет захватывать специальные формы в том числе, а во-вторых, он выполняет это статически, а не динамически.

Также рассмотренная макросистема позволяет создание предопределённых макросов, чьё написание невозможно для конечного пользователя; например, можно определить макрос, создающий глобальные переменные прямым вызовом функции `insert-global!`.

Несмотря на это, она имеет и недостатки. Если пользователь захочет посмотреть, во что именно раскрываются макросы, то он столкнётся как минимум с двумя затруднениями. Во-первых, ему недоступна функция `expand`. Очевидно, это можно исправить, сделав так, чтобы переменная `expand` текущего уровня интерпретации ссылалась на функцию `expand` следующего. Вторая проблема заключается в том, что результатом раскрытия является объект класса `Program`, чья структура неизвестна пользователю. Но и это тоже поправимо: [**см. упр. 9.4**].

Одной из задач данной макросистемы было продемонстрировать тесную связь между макросами и компиляцией — оба понятия имеют поразительно много общего. Если из приведённой реализации убрать код, ответственный за собственно вычисления и объектификацию, то о макросах в нём будет напоминать только небольшая часть функций `objectify` и `objectify-symbol`, а также четыре обработчика предопределённых макросов. Суммарно это чуть меньше сотни строк кода — очень мало.

Серьёзная макросистема, конечно же, должна включать в себя гораздо больше полезных вещей:

1) встроенные синтаксические возможности вроде `or`, `letrec`, вложенных `define` и т. д.;

2) квазицитирование, органично объединённое с объектификацией и раскрытием макросов;

3) пользовательские макросимволы, позволяющие удобное написание программ в стиле `define-handy-method`. [**см. стр.** 405]

Тем не менее, одной из изначальных целей — ознакомиться с идеей захвата смысла символов — мы всё же достигли.

9.12. Заключение

Все проблемы, связанные с макросами, можно обобщить двумя словами: необходимость и несовместимость. Макросы регулярно используются на практике в том или ином виде, но при этом существует огромное число несовместимых друг с другом реализаций макросистем, предоставляющих различные возможности. В этой главе мы попробовали охватить максимально широкий ассортимент решений. Немногие руководства по реализациям Лиспа и Scheme точно описывают используемую модель макросов, однако в их защиту можно сказать, что данная глава, вероятно, является одной из первых попыток описать и классифицировать безмерное множество вариантов поведения макросистем.

9.13. Упражнения

Упражнение 9.1 Определите макрос `repeat`, который был описан в начале этой главы. Сделайте его гигиеничным.

Упражнение 9.2 С помощью `define-syntax` определите макрос, принимающий последовательность выражений и печатающий их по порядку рядом с их номерами. Например, макровызов `(enumerate` π_0 π_1 `...` π_n`)` должен раскрыться в нечто, что при выполнении напечатает `0`, затем значение π_0, потом `1`, а за ней значение π_1, и так далее.

Упражнение 9.3 Перенесите макросистему из данной главы в истинно единый мир.

Упражнение 9.4 Напишите функцию, преобразующую объекты `Program` обратно в эквивалентные S-выражения.

Упражнение 9.5 Изучите реализацию MEROONET [**см. стр.** 494] и определите, что оттуда относится к библиотеке времени раскрытия, что к библиотеке времени исполнения, а что к обеим.

Рекомендуемая литература

Весьма интересные мысли в защиту макросов приведены в книге [Gra93]. Есть также и работы больше теоретического характера, вроде [QP90]. О проблемах гигиеничного макрораскрытия можно почитать в [KFFD86, DFH88, CR91a, QP91b]. Наконец, стоит обратить внимание на многообещающую модель раскрытия, рассмотренную в работе [DM95].

Глава 10

Компиляция в Си

Ещё одна глава о компиляции. Но в этот раз мы разберём несколько новых подходов, в частности, плоские окружения, а ещё у нас сменился целевой язык: теперь им будет Си. Подобное сочетание языков поднимает несколько своеобразных вопросов, стоящих рассмотрения. С одной стороны, брак с представителем более высокого сословия сулит определённые выгоды: например, бесплатные низкоуровневые оптимизации и доступ к огромному количеству библиотек на все случаи жизни. Но роз без шипов не бывает, так что взамен мы отдаём хвостовую рекурсию и вдобавок получаем кучу проблем со сборкой мусора.

Компиляция в язык высокого уровня, вроде Си, интересна во многих отношениях. Так как целевой язык несомненно богаче ассемблера, то всё же можно надеяться на несколько более близкий к оригиналу результат, нежели бесформенный ералаш байт-кодов. Далее, компиляторы Си существуют практически для любой платформы, что благотворно сказывается на переносимости. Более того, эти компиляторы не вчера родились и имеют в своём распоряжении существенный арсенал оптимизаций, а также неплохо справляются с рутинными низкоуровневыми задачами вроде распределения регистров, расположения данных в памяти, выбора режимов адресации — обо всём этом можно забыть, сфокусировавшись исключительно на высокоуровневом аспекте компиляции.

С другой стороны, выбор языка высокого уровня в качестве целевого накладывает определённые философские и практические ограничения. Подобные языки обычно разрабатываются под определённый стиль программирования и, чаще всего, без особого внимания к вопросу кодогенерации. В результате обнаруживаются разнообразные «стеклянные стены» вроде не более 32 аргументов у функций, или 16 максимальных уровней вложенности блоков, и так далее. Нормальные пользователи их даже не замечают, а вот глупые программы-генераторы то и дело натыкаются. Бывает, что какой-то жалкий макрос раздувает программу в 5, 10, а то и в 20 раз при переводе на Си, а потом этого монстра ещё пытаются скормить компилятору.

Более того, если модель исполнения целевого языка имеет мало общего с моделью языка исходного, это плохо сказывается как на сложности самого процесса преобразования, так и на эффективности использования возможно-

стей целевого языка. Си разрабатывался как язык системного программирования (изначально для написания Un⋆x), поэтому в нём намеренно применяется явное управление памятью. Вследствие этого могут возникать невероятно каверзные ошибки, если программист не будет предельно осторожен с указателями, — иначе это лишь вопрос времени, когда программа выйдет из-под контроля и начнёт творить непоправимое. В Лиспе подобная ситуация невозможна в принципе, это безопасный язык в плане управления памятью.

Вдобавок ко всему, Си не особо подходит для функционального стиля программирования, где вызовы функций и рекурсия случаются на каждом шагу. Компиляторы тщательнее оптимизируют императивные аспекты языка, поэтому программисты стараются избегать менее эффективных в данном случае функциональных алгоритмов, что даёт разработчикам компиляторов моральное право уделять меньше внимания их оптимизации, так как они всё равно редко используются. Такой вот замкнутый круг.

Как бы то ни было, нельзя сказать, что Си непопулярен в качестве целевого языка компиляции Лиспа. Достаточно упомянуть Kyoto Common Lisp [YH85] (а также улучшенный Уильямом Скэлтером AKCL), WCL [Hen92b], CLICC [Hof93], ECoLisp [Att95], Scheme→C [Bar89], Sqil [Sén91], ILOG Talk [ILO94], Bigloo [Ser94].

Наш компилятор им не соперник. Это лишь набросок, однако имеющий достаточное количество интересных деталей. Конечно, можно было бы пойти простым путём (кто-то поправит: тривиальным) — взять компилятор в байткод из седьмой главы [**см. стр. 269**] и переработать его, чтобы он выводил код на Си, а не байты. Ведь для каждой машинной инструкции определена её операционная семантика, так что не составит труда перевести их все на Си. Однако нас ожидает совершенно иной путь, пролегающий сквозь страну плоских окружений. [**см. стр. 243**] Рассматриваемый компилятор будет многопроходным. Объектное представление обрабатываемых программ значительно облегчит их анализ и трансформацию. По сути, именно систематический объектно-ориентированный подход делает излагаемый в данной главе материал столь элегантным и расширяемым.

10.1. Объектификация

В разделе 9.11.1 [**см. стр. 410**] рассматривался алгоритм преобразования программ в объектную форму. Он раскрывает все макросы и в конечном итоге возвращает готовый объект класса `Program`. Чтобы разбавить это весьма лаконичное описание, вашему вниманию предлагается рисунок 10.1 с результатом объектификации следующей далее программы. В ней, если присмотреться, можно найти каждый изученный нами существенный аспект Scheme. Страницы не безграничны, так что имена классов записаны сокращённо, а часть дерева и вовсе не показана. На протяжении этой главы мы не раз будем возвращаться к данному примеру.

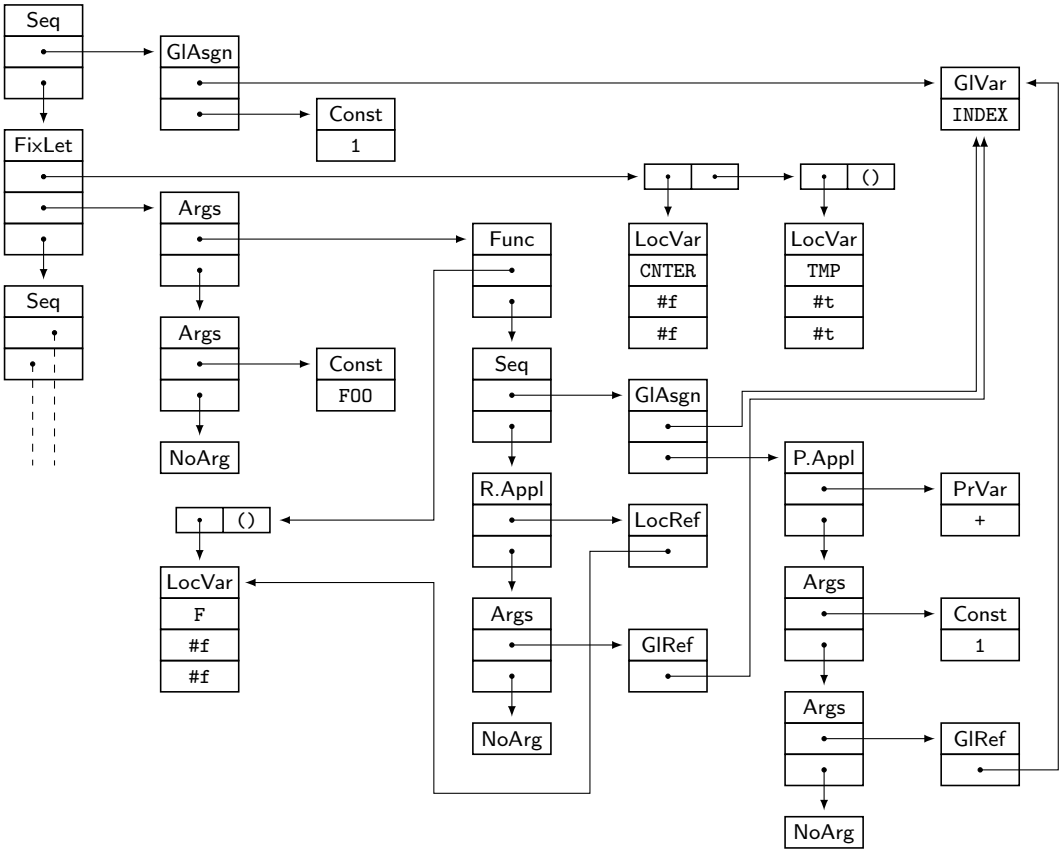


Рис. 10.1. Объектифицированный код.

```
(begin
  (set! index 1)
  ((lambda (cnter . tmp)
     (set! tmp (cnter (lambda (i) (lambda x (cons i x)))))
     (if cnter (cnter tmp) index) )
   (lambda (f)
     (set! index (+ 1 index))
     (f index) )
   'foo) )          → (2 3)
```

10.2. Обход кода

Обход кода является чрезвычайно важным приёмом при построении компиляторов. [Cur89, Wat93] Суть его состоит в обходе построенного ранее дерева, представляющего обрабатываемую программу. В процессе него собирается информация, в узлах дерева оставляются разнообразные пометки, и так далее; всё это делается ради облегчения следующего этапа — кодогенерации. Порядок обхода, а также используемые структуры данных сильно зависят от

поставленных целей, поэтому какого-либо универсального алгоритма не существует. Если вдуматься, то все наши интерпретаторы — это именно обходчики кода, которые исполняют его в процессе; и предобработчик `meaning` из главы про быструю интерпретацию [**см. стр. 222**] также является специализированным обходчиком.

Мы будем использовать всего один способ обхода дерева, но выполняемые в это время действия могут быть различными. Для такого случая отлично подходит идиома *метаметода* (или метода высшего порядка, если хотите). Функция `update-walk!` принимает следующие аргументы: обобщённую функцию g, объект o класса `Program`, а также неопределённое количество дополнительных аргументов для g. Для каждого поля f объекта o, содержащего экземпляр `Program`, она заменяет его значение результатом аппликации $(g\ f)$. Возвращаемым значением `update-walk!` является обновлённый объект. Надеюсь, теперь вам стало понятно, зачем класс `Arguments`, представляющий аргументы аппликаций, сделан наследником `Program`: ведь аргументы тоже вычисляются и их надо исследовать в процессе обхода.

```
(define (update-walk! g o . args)
  (for-each (lambda (field)
              (let ((vf (field-value o field)))
                (when (Program? vf)
                  (let ((v (if (null? args) (g vf)
                               (apply g vf args) )))
                    (set-field-value! o v field) ) ) ) )
            (Class-fields (object->class o)) )
  o )
```

Может показаться, что такой подход чересчур уж всё упрощает, но это действительно весьма удобная функция для преобразования программ, как вы сами убедитесь в дальнейшем. Очевидно, что некоторые этапы различных трансформаций возможно объединить и выполнить за один обход дерева, ускорив тем самым обработку. Однако, мы оградим себя от соблазна во имя чёткого разделения эффектов применяемых трансформаций.

10.3. Пакуем коробки

Для начала давайте заменим все локальные присваивания операциями над коробками. Подобное преобразование уже разбиралось ранее [**см. стр. 145**], поэтому оно будет хорошим примером использования `update-walk!`.

Итак, каждое присваивание локальной переменной должно быть изменено на помещение соответствующего значения в коробку. Естественно, обращения к переменным тоже преобразуются в извлечения значений из соответствующих коробок. Интерфейсом к обходчику кода служат обобщённые функции, так что приведённые выше преобразования надо выразить в виде одной из них. Как вы помните, коробочное преобразование выполнялось рекурсивно для

всего выражения — точно так же должна действовать и обобщённая функция. Её совместная работа с обходчиком — это залог успеха и основа могущества данного союза.

```
(define-generic (insert-box! (o Program))
  (update-walk! insert-box! o) )
```

Мы введём три новых класса узлов синтаксического дерева для представления новых понятий, необходимых при работе с коробками. Подробно каждый из классов будет рассмотрен позже, вместе с соответствующими этапами преобразования.

```
(define-class Box-Read     Program (reference))
(define-class Box-Write    Program (reference form))
(define-class Box-Creation Program (variable))
```

Благодаря удачному решению помечать во время объектификации изменяемые локальные переменные флагом `mutable?`, заменить обращения к подобным переменным вызовами `Box-Read` становится элементарно:

```
(define-method (insert-box! (o Local-Reference))
  (if (Local-Variable-mutable? (Local-Reference-variable o))
      (make-Box-Read o)
      o ) )
```

Преобразование присваиваний сравнимо по сложности. Присваивание локальной переменной однозначно переводится в `Box-Write`. Нужно только не забыть о рекурсивной природе преобразования и вызвать обходчик для выражения, вычисляющего новое значение переменной.

```
(define-method (insert-box! (o Local-Assignment))
  (make-Box-Write (Local-Assignment-reference o)
                  (insert-box! (Local-Assignment-form o)) ) )
```

В данный момент программа считает, что все изменяемые переменные уже разложены по коробкам; нам остаётся лишь оправдать её ожидания. Локальные изменяемые переменные создаются исключительно `lambda`- и `let`-формами, то есть узлами классов `Function` и `Fix-Let`.[1] Идея состоит в том, чтобы поместить создание коробок перед телом этих форм. [**см. стр. 146**] Просто и понятно, так что приступаем к написанию соответствующих специализаций функции `insert-box!`. Каждая из них заказывает у вспомогательной функции необходимое количество коробок, после чего вставляет полученные экземпляры `Box-Creation` куда следует.

```
(define-method (insert-box! (o Function))
  (set-Function-body!
   o (insert-box!
      (boxify-mutable-variables (Function-body o)
                                (Function-variables o) ) ) )
  o )
```

[1] Даже если сейчас в дереве нет `Fix-Let`-узлов, соответствующих приводимым формам, метод их обработки должен существовать.

```
(define-method (insert-box! (o Fix-Let))
  (set-Fix-Let-arguments! o (insert-box! (Fix-Let-arguments o)))
  (set-Fix-Let-body!
   o (insert-box!
      (boxify-mutable-variables (Fix-Let-body o)
                                (Fix-Let-variables o) ) ) )
  o )

(define (boxify-mutable-variables form variables)
  (if (pair? variables)
      (if (Local-Variable-mutable? (car variables))
          (make-Sequence
           (make-Box-Creation (car variables))
           (boxify-mutable-variables form (cdr variables)) )
          (boxify-mutable-variables form (cdr variables)) )
      form ) )
```

Готово. Мы полностью определили коробочное преобразование с помощью всего лишь четырёх предельно понятных методов. Результат его выполнения можно увидеть на рисунке 10.2 (показана только изменившаяся часть).

10.4. Избавляемся от вложенных функций

Язык Си не позволяет определять функции внутри других функций. Иными словами, вложенные `lambda`-формы нельзя прямо перевести на Си. Следовательно, от них необходимо избавиться, преобразовав программу в набор комбинаторов — функций без свободных переменных. К счастью, это довольно известное преобразование, называемое *λ-поднятием* (λ-lifting) — в результате него `lambda`-формы поднимаются вверх, до самой поверхности, не оставляя ни одной свободной переменной в глубине программы. Существует множество вариантов реализации подобного преобразования, по-разному обращающихся с исходными функциями и сохраняющих те или иные их аспекты, например: [WS94, KH89, CH94].

Результатом вычисления любой `lambda`-формы является замыкание — сущность, сохраняющая в себе окружение, где она была создана. При вызове замыкания специальная функция (известная как `invoke`) берёт на себя работу по организации вычисления тела данного замыкания в правильном окружении, составленном из аргументов функции, находящихся в окружении вызова, и свободных переменных, извлекаемых из замкнутого окружения. Фактически, это единственная функция, которой известна истинная природа замыканий, поэтому мы можем довольно легко изменить их структуру, не потревожив остальные части программы: ведь каждая аппликация в конечном итоге выполняется `invoke`.

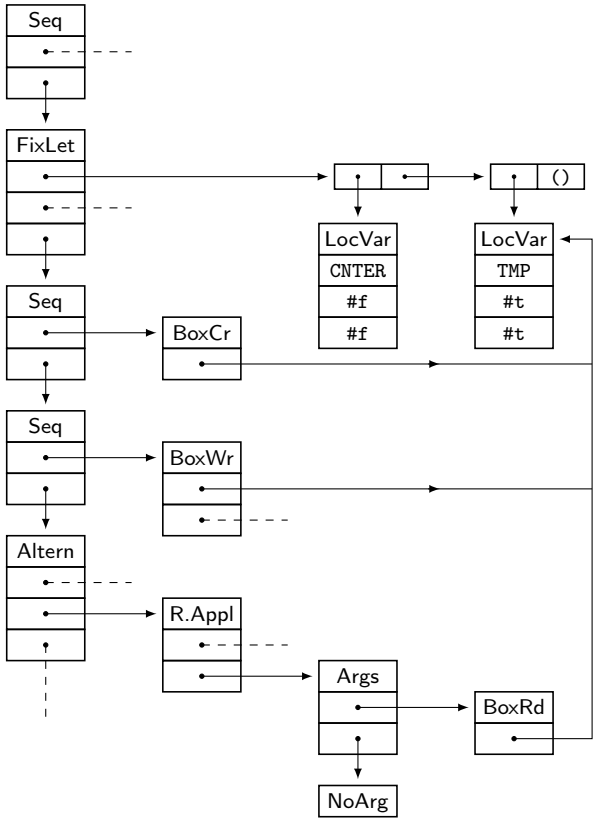


Рис. 10.2. `(lambda (cnter . tmp) (set! tmp ...) (if ... (... tmp) ...))`.

Рассмотрим упомянутое преобразование на примере *ОО-подъёма*, описанного в [Que94]. Как обычно, морской свинкой будет факториал:

```
(define (fact n k)
  (if (= n 0) (k 1)
      (fact (- n 1) (lambda (r) (k (* n r))) ) ) )
```

От `lambda`-формы со свободными переменными `k` и `n` можно избавиться следующим образом:

```
(define-class Fact-Closure Object (n k))

(define-method (invoke (f Fact-Closure) r)
  (invoke (Fact-Closure-k f) (* (Fact-Closure-n f) r)) )

(define (fact n k)
  (if (= n 0) (invoke k 1)
      (fact (- n 1) (make-Fact-Closure n k)) ) )
```

Суть преобразования состоит в замене неявной структуры создаваемого анонимного замыкания явным объектом класса `Fact-Closure`, содержащим

все необходимые свободные переменные (здесь это `n` и `k`). После этого остаётся только научить `invoke` корректным образом обрабатывать подобные объекты.

Для выполнения такого преобразования вначале надо как-то получить и где-то сохранить список свободных переменных обрабатываемых функций. Для этого введём новый класс преобразованных функций — `Flat-Function`, а их свободные переменные будем собирать в списки из `Free-Environment`, заканчивающиеся на `No-Free`. Как и их братья `Arguments` и `No-Argument`, эти два класса являются наследниками `Program`, потому что представляют вычислимые понятия. Обращения к свободным переменным тоже будут преобразованы в специальные объекты класса `Free-Reference`.

```
(define-class Flat-Function    Function  (free))
(define-class Free-Environment Program   (first others))
(define-class No-Free          Program   ())
(define-class Free-Reference   Reference ())
```

Реализуется это преобразование, естественно, с помощью обходчика кода. Функция `lift!` будет служить удобным интерфейсом, а всю чёрную работу мы переложим на обобщённую функцию `lift-procedures!`. Помимо обрабатываемой функции, она принимает два вспомогательных аргумента: абстракцию `flatfun`, которая в конечном итоге будет содержать все свободные переменные, и список `vars`, где лежат все переменные, который на данный момент связаны. По умолчанию `lift-procedures!` просто рекурсивно вызывается для всех дочерних узлов обрабатываемой программы.

```
(define (lift! o)
  (lift-procedures! o #f '()) )

(define-generic (lift-procedures! (o Program) flatfun vars)
  (update-walk! lift-procedures! o flatfun vars) )
```

Преобразование определяется всего тремя методами. Первый из них отвечает за обработку обращений к свободным переменным и поддержание списка `flatfun` в адекватном состоянии. Функция `adjoin-free-variable!` помогает с занесением новых свободных переменных в этот список.

```
(define-method (lift-procedures! (o Local-Reference) flatfun vars)
  (let ((v (Local-Reference-variable o)))
    (if (memq v vars) o
        (begin (adjoin-free-variable! flatfun o)
               (make-Free-Reference v) ) ) ) )

(define (adjoin-free-variable! flatfun ref)
  (when (Flat-Function? flatfun)
    (let check ((free* (Flat-Function-free flatfun)))
      (if (No-Free? free*)
          (set-Flat-Function-free!
           flatfun (make-Free-Environment
                    ref (Flat-Function-free flatfun) ) )
```

```
          (unless (eq? (Reference-variable ref)
                       (Reference-variable
                        (Free-Environment-first free*) ) )
            (check (Free-Environment-others free*)) ) ) ) ) )
```

Как известно, форма `let` создаёт локальные привязки, поэтому перед обработкой тела `Fix-Let` необходимо поместить все свежесозданные связанные переменные в список `vars`. Ещё одним неоспоримым фактом является то, что приводимые формы, к которым относится `let`, лишь вводят новые переменные и не требуют создания замыканий. Это очень важная оптимизация, и мы отнюдь не хотим её лишиться.

```
(define-method (lift-procedures! (o Fix-Let) flatfun vars)
  (set-Fix-Let-arguments!
   o (lift-procedures! (Fix-Let-arguments o) flatfun vars) )
  (let ((newvars (append (Fix-Let-variables o) vars)))
    (set-Fix-Let-body!
     o (lift-procedures! (Fix-Let-body o) flatfun newvars) )
    o ) )
```

Наконец, остался самый сложный случай — абстракции. Тело абстракции анализируется и все её свободные переменные собираются в объекте `Flat-Function`. Так как свободные переменные вложенной абстракции сами могут оказаться свободными в содержащей её абстракции, то для них обходчик вызывается ещё раз, чтобы протолкнуть их как можно выше. Сейчас самое время порадоваться принятому ранее решению сделать список свободных переменных наследником `Program`.

```
(define-method (lift-procedures! (o Function) flatfun vars)
  (let* ((localvars (Function-variables o))
         (body      (Function-body o))
         (newfun (make-Flat-Function localvars body (make-No-Free))) )
    (set-Flat-Function-body!
     newfun (lift-procedures! body newfun localvars) )
    (let ((free* (Flat-Function-free newfun)))
      (set-Flat-Function-free!
       newfun (lift-procedures! free* flatfun vars) ) )
    newfun ) )
```

Как полагается, на рисунке 10.3 показан результат применения рассмотренного преобразования к нашему бессменному примеру.

10.5. Собираем цитаты и функции

Предыдущее преобразование не изменяло расположения функций: вложенные `lambda`-формы так и остались вложенными, пусть и без свободных переменных. Нет, мы не забыли об этом! Просто немного задержались, чтобы выполнить перенос вместе со следующим преобразованием — извлечением цитат.

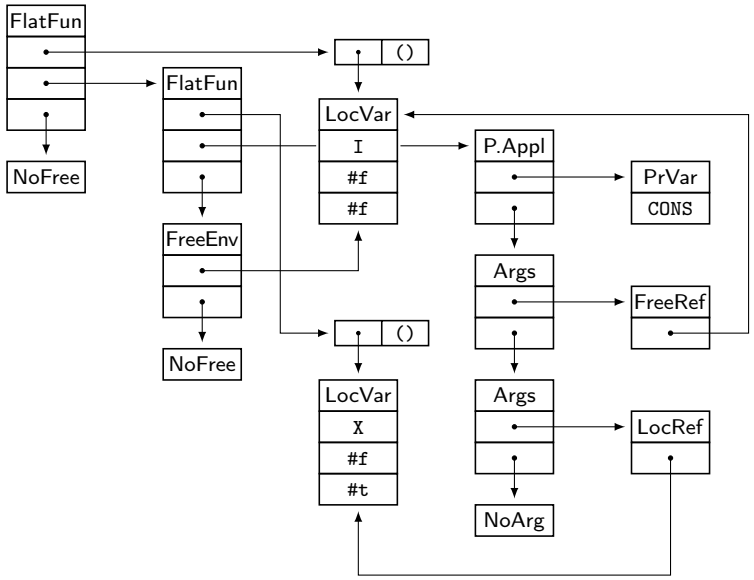


Рис. 10.3. `(lambda (i) (lambda x (cons i x)))`.

Во время следующего обхода кода мы извлечём все используемые цитаты и определения функций, чтобы перенести их на глобальный уровень, как этого требует Си. Здесь понадобятся всего два метода.

Функция `extract-things!` преобразовывает объекты `Program` в объекты `Flattened-Program`. Новый класс является наследником `Program` и имеет три дополнительных поля: `form` для собственно программы, `quotations` для списка цитат и `definitions` для определений функций. До значений цитат можно добраться через глобальные переменные нового класса `Quotation-Variable`. Функции мы просто пронумеруем, выдав каждой из них уникальный `index`. Наконец, создание замыканий перекладывается на новый тип узлов синтаксического дерева — `Closure-Creation`. (Не волнуйтесь, скоро вам станет ясно, почему всё сделано именно так.)

```
(define-class Flattened-Program   Program (form quotations definitions))
(define-class Quotation-Variable  Variable (value))
(define-class Function-Definition Flat-Function (index))
(define-class Closure-Creation    Program (index variables free))
```

Строго говоря, `extract-things!` работает не одна, а вместе с обобщённой функцией `extract!`. Взаимодействие подразумевает обмен информацией. Дабы не прибегать для этого к глобальным переменным (например, ради распараллеливания компиляции), мы будем возвращать результат через специальный аргумент, передаваемый обобщённой функции.

```
(define (extract-things! o)
  (let ((result (make-Flattened-Program o '() '())))
    (set-Flattened-Program-form! result (extract! o result))
    result ) )
```

```
(define-generic (extract! (o Program) result)
  (update-walk! extract! o result) )
```

Цитаты просто собираются в отведённом для них поле, а все обращения к ним заменяются обращениями к соответствующим глобальным переменным, инициализированным значениями исходных цитат.

```
(define-method (extract! (o Constant) result)
  (let* ((qv* (Flattened-Program-quotations result))
         (qv  (make-Quotation-Variable (length qv*)
                                       (Constant-value o) )) )
    (set-Flattened-Program-quotations! result (cons qv qv*))
    (make-Global-Reference qv) ) )
```

Абстракции находятся в узлах типа `Flat-Function`, которые преобразуются в экземпляры `Closure-Creation`. Процесс можно было бы оптимизировать, избежав дублирования кода для одинаковых абстракций, научив `adjoin-definition!` отыскивать повторения. Также может показаться странным, что столько внимания уделяется спискам свободных переменных изначальных абстракций. Это делается исключительно для облегчения определения арности функций во время генерации Си-кода.

```
(define-method (extract! (o Flat-Function) result)
  (let* ((newbody   (extract! (Flat-Function-body o) result))
         (variables (Flat-Function-variables o))
         (freevars  (let extract ((free (Flat-Function-free o)))
                      (if (Free-Environment? free)
                          (cons (Reference-variable
                                 (Free-Environment-first free) )
                                (extract
                                 (Free-Environment-others free) ) )
                          '() ) ))
         (index (adjoin-definition!
                 result variables newbody freevars )) )
    (make-Closure-Creation index variables (Flat-Function-free o)) ) )

(define (adjoin-definition! result variables body free)
  (let* ((definitions (Flattened-Program-definitions result))
         (newindex (length definitions)) )
    (set-Flattened-Program-definitions!
     result (cons (make-Function-Definition
                   variables body free newindex )
                  definitions ) )
    newindex ) )
```

И снова, на рисунке 10.4 показан результат применения преобразования к иной части примера.

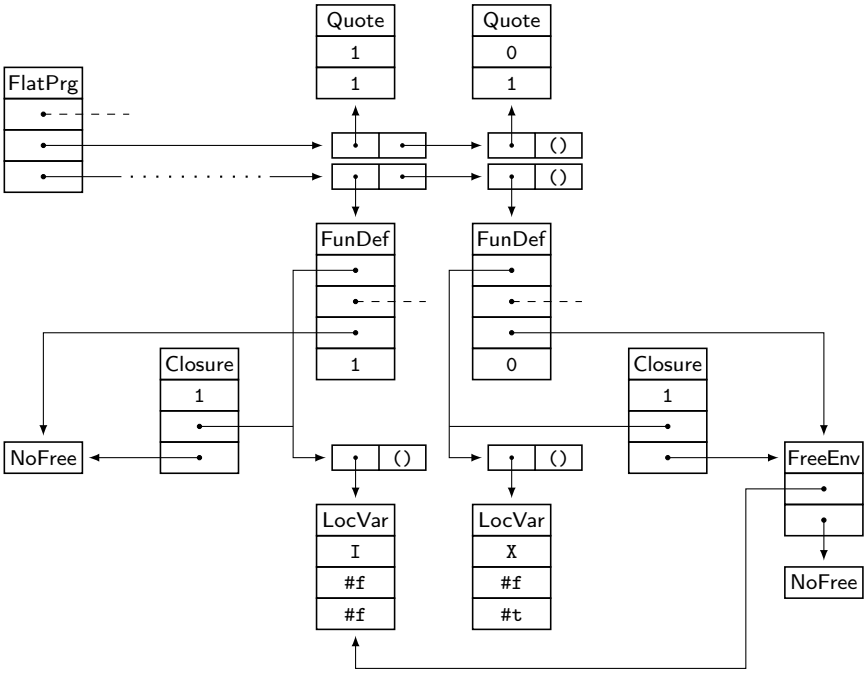


Рис. 10.4. `(begin (set! index 1) ((lambda ...) ... ))`.

Наконец, вся программа превращается в вызов одной огромной функции. То есть π становится `((lambda () `π`))`.

```
(define (closurize-main! o)
  (let ((index (length (Flattened-Program-definitions o))))
    (set-Flattened-Program-definitions!
     o (cons (make-Function-Definition
              '() (Flattened-Program-form o) '() index )
             (Flattened-Program-definitions o) ) )
    (set-Flattened-Program-form!
     o (make-Regular-Application
        (make-Closure-Creation index '() (make-No-Free))
        (make-No-Argument) ) )
    o ) )
```

10.6. Собираем временные переменные

У автора всегда есть решительное преимущество перед читателем в том, что он знает, что хочет получить, а вы пока ещё не знаете, что получится в результате. В надежде на оригинальность, выражения Scheme было решено переводить в выражения Си. Это выглядит немного эксцентричным, так как язык Си вообще-то построен вокруг инструкций (statements), а не выражений (expressions), но такое представление позволяет сохранить структуру Scheme

в получаемой на выходе программе. Правда, в этом случае возникает проблема с переводом узлов типа `Fix-Let`, так как в языке Си нельзя создавать локальные переменные внутри выражений,[2] только с помощью отдельных инструкций. Поэтому мы вынуждены будем пробежаться по всем `Fix-Let` и провести учёт всех локальных переменных, вводимых ими, чтобы впоследствии корректно их объявить.

Естественно, для этого выполняется ещё один проход по коду. Задачей данного прохода будет сбор всех локальных переменных форм `Fix-Let` в одном месте, а также их переименование для предотвращения коллизий. Функции, имеющие временные локальные переменные, мы будем представлять подклассом `Function-Definition`, называемым `With-Temp-Function-Definition`.

```
(define-class With-Temp-Function-Definition Function-Definition
  ( temporaries ) )
```

Функция `gather-temporaries!` реализует преобразование. Она полагается на обобщённую функцию `collect-temporaries!`, работающую в паре с обходчиком. Вторым аргументом данной функции является объект, куда она будет складывать обнаруженные переменные, а третий хранит список соответствий старых имён новым, необходимый для выполнения переименований.

```
(define (gather-temporaries! o)
  (set-Flattened-Program-definitions!
   o (map (lambda (def)
            (let ((flatfun (make-With-Temp-Function-Definition
                            (Function-Definition-variables def)
                            (Function-Definition-body def)
                            (Function-Definition-free def)
                            (Function-Definition-index def)
                            '() )))
              (collect-temporaries! flatfun flatfun '()) ) )
          (Flattened-Program-definitions o) ) )
  o )

(define-generic (collect-temporaries! (o Program) flatfun r)
  (update-walk! collect-temporaries! o flatfun r) )
```

Для завершения преобразования остаётся определить три новых метода. Локальные переменные при необходимости могут переименовываться, то же самое касается и переменных в коробках:

```
(define-method (collect-temporaries! (o Local-Reference) flatfun r)
  (let* ((variable (Local-Reference-variable o))
         (v (assq variable r)) )
    (if (pair? v) (make-Local-Reference (cdr v))
                  o ) ) )
```

[2] Однако, например, `gcc` расширяет язык формой `({ ... })`, позволяющей вводить переменные где угодно.

```
(define-method (collect-temporaries! (o Box-Creation) flatfun r)
  (let* ((variable (Box-Creation-variable o))
         (v (assq variable r)) )
    (if (pair? v) (make-Box-Creation (cdr v))
                  o ) ) )
```

Самый сложный метод, конечно же, будет для `Fix-Let`. Сначала он рекурсивно вызывается для всех её аргументов, после чего переименовывает локальные переменные с помощью функции `new-renamed-variable`. Затем эти новые переменные добавляются в определение обрабатываемой формы, а `collect-temporaries!` ещё раз рекурсивно вызывается уже для её тела, помещённого в обновлённое окружение.

```
(define-method (collect-temporaries! (o Fix-Let) flatfun r)
  (set-Fix-Let-arguments!
   o (collect-temporaries! (Fix-Let-arguments o) flatfun r) )
  (let* ((newvars (map new-renamed-variable (Fix-Let-variables o)))
         (newr (append (map cons (Fix-Let-variables o) newvars) r)) )
    (adjoin-temporary-variables! flatfun newvars)
    (set-Fix-Let-variables! o newvars)
    (set-Fix-Let-body!
     o (collect-temporaries! (Fix-Let-body o) flatfun newr) )
    o ) )

(define (adjoin-temporary-variables! flatfun newvars)
  (let adjoin ((temps (With-Temp-Function-Definition-temporaries
                       flatfun ))
               (vars newvars) )
    (if (pair? vars)
        (if (memq (car vars) temps)
            (adjoin temps (cdr vars))
            (adjoin (cons (car vars) temps) (cdr vars)) )
        (set-With-Temp-Function-Definition-temporaries!
         flatfun temps ) ) ) )
```

Переименованные переменные выделяются в собственный подкласс. Ещё нам потребуется счётчик для обеспечения уникальности их имён.

```
(define-class Renamed-Local-Variable Variable (index))

(define renaming-variable-counter 0)

(define-generic (new-renamed-variable (variable)))

(define-method (new-renamed-variable (variable Local-Variable))
  (set! renaming-variable-counter (+ 1 renaming-variable-counter))
  (make-Renamed-Local-Variable
   (Variable-name variable) renaming-variable-counter ) )
```

10.7. Передышка

Следующая программа на Scheme является финальным результатом метаморфоз, произошедших с показанным в самом начале главы примером. Итого:

1) Изменяемые переменные разложены по коробками.

2) Вложенные функции не используются.

3) Все цитаты и определения функций вынесены наверх.

4) Временные переменные создаются явно.

```
(define quote_5 'foo)                    ; собранные цитаты
(define-class Closure_0 Object ())       ; абстракция (lambda (f) ...)
(define-method (invoke (self Closure_0) f)
  (begin
    (set! index (+ 1 index))
    (invoke f index) ) )                 ; вызов функции
(define-class Closure_1 Object (i))      ; абстракция (lambda (x) ...)
(define-method (invoke (self Closure_1) . x)
  (cons (Closure_1-i self)               ; свободная переменная i
        x ) )
(define-class Closure_2 Object ())       ; абстракция (lambda (i) ...)
(define-method (invoke (self Closure_2) i)
  (make-Closure_1 i) )                   ; создание замыкания
(define-class Closure_3 Object ())       ; абстракция (lambda () π)
(define-method (invoke (self Closure_3))
  ((lambda (cnter_1 tmp_2)               ; переименование в действии
     (set! index 1)
     (set! cnter_1 (make-Closure_0))     ; инициализация
     (set! tmp_2 (cons quote_5 '()))     ;   временных переменных
     (set! tmp_2 (make-box tmp_2))       ; укладываем в коробку
     (box-write! tmp_2                   ;   изменяемую переменную
                 (invoke cnter_1 (make-Closure_2)) )
     (if cnter_1
         (invoke cnter_1 (box-read tmp_2))
         index ) ) ) )
(invoke (make-Closure_3))                ; точка входа
```

10.8. Генерируем Си

Наконец, мы готовы выполнить мистический обряд кодогенерации. Осталось лишь дать несколько предварительных пояснений. Ваш покорный слуга не мнит себя мировым экспертом по языку Си и, в сущности, обязан своими

знаниями тщательному изучению книг вроде [ISO90, HS91, CEK+89]. Предполагается, что вы имеете некоторое представление об этом языке, но не обременены предрассудками о единственно верном способе его использования.

Абстрактное синтаксическое дерево полностью готово и только и ждёт, чтобы его скомпилировали, а вернее сказать, *распечатали* на языке Си. Благодаря высокоуровневому представлению программ, генерация кода не представляет особых сложностей. Функция `compile->C` принимает S-выражение, выполняет над ним рассмотренные ранее преобразования и в конце концов выводит в порт `out` опрятный исходный код на Си.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things! (lift! (Sexp->object e)))))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )

(define (generate-C-program out e prg)
  (generate-header             out e)
  (generate-global-environment out g.current)
  (generate-quotations         out (Flattened-Program-quotations prg))
  (generate-functions          out (Flattened-Program-definitions prg))
  (generate-main               out (Flattened-Program-form prg))
  (generate-trailer            out)
  prg )
```

Как и любая программа на Си, а в более широком смысле, как и любое животное, получаемая программа состоит из головы, тела и хвоста. В голове находятся желания, умения и мысли, поэтому там размещается комментарий с исходным выражением на Scheme (выводимым нестандартной функцией `pp`), а также директива препроцессора, подключающая к программе заголовочный файл `scheme.h`, где будут находиться все необходимые ей определения. Для вывода строк программы здесь и далее используется ещё одна нестандартная функция `format`.

```
(define (generate-header out e)
  (format out "/* Compiler to C $Revision: 4.1$~%")
  (pp e out)
  (format out "~%*/~%~%#include \"scheme.h\"~%") )

(define (generate-trailer out)
  (format out "~%/* End of generated code */~%") )
```

В хвосте, как видите, нет ничего интересного; всё самое важное находится в теле программы. Результат компиляции нашего подопытного примера приведён на странице 460. Возможно, вам захочется взглянуть на него, прежде чем переходить к изучению анатомии генератора.

10.8.1. Глобальное окружение

Глобальные переменные, используемые программой, можно разделить на две группы: предопределённые переменные, вроде `car` или `+`, и изменяемые глобальные переменные. Считается, что предопределённые переменные неизменяемы, а их значения физически располагаются в специальной библиотеке, которая будет подключена при компоновке. В отличие от них, окружение глобальных изменяемых переменных должно быть сформировано именно у нас в программе. Для этого мы используем информацию, которую собрали в `g.current` во время предобработки.

Используемый подход к генерации строится на предположении, что программа компилируется целиком, а не отдельными модулями. (Раздельная компиляция, как вы помните, вызывает множество специфичных затруднений. [**см. стр. 314**] Мы не будем их рассматривать повторно, дабы не раздувать эту главу в пару раз.)

Для облегчения понимания и повышения читабельности кода, генератор щедро использует макросы Си. Например, глобальные переменные объявляются макросом `SCM_DefineGlobalVariable`. Вторым аргументом этот макрос принимает строку с изначальным именем[3] переменной на Scheme. Это может оказаться полезным при отладке.

```
(define (generate-global-environment out gv*)
  (when (pair? gv*)
    (format out "~%/* Global environment: */~%")
    (for-each (lambda (gv) (generate-global-variable out gv))
              gv* ) ) )

(define (generate-global-variable out gv)
  (let ((name (Global-Variable-name gv)))
    (format out "SCM_DefineGlobalVariable(~A,\"~A\");~%"
            (IdScheme->IdC name) name ) ) )
```

Вот мы и встретились с первым затруднением: не все идентификаторы Scheme являются допустимыми в Си. Функция `IdScheme->IdC` занимается преобразованием идентификаторов Scheme в корректные идентификаторы Си. Она должна избавиться от всех мешающих символов, оставив при этом имя более-менее понятным, чтобы можно было догадаться о его исходном написании. Естественно, вариантов такого преобразования огромное множество, но мы просто заменим все недопустимые символы допустимыми, после чего убедимся в уникальности полученного имени. Код, который это делает, не то, чтобы особо интересен, но в самом начале книги у нас был уговор: ничего не утаивать. Переменная `Scheme->C-names-mapping` хранит словарик имён, изначально содержащий несколько сложных случаев.

[3] Используемая в этой главе функция `read` переводит все имена символов в верхний регистр.

```
(define Scheme->C-names-mapping
  '( (*        . "TIMES")
     (<        . "LESSP")
     (pair?    . "CONSP")
     (set-cdr! . "RPLACD")
     ; и другие
     ) )
(define (IdScheme->IdC name)
  (let ((v (assq name Scheme->C-names-mapping)))
    (if (pair? v) (cdr v)
        (let ((str (symbol->string name)))
          (let retry ((Cname (compute-Cname str)))
            (if (Cname-clash? Cname Scheme->C-names-mapping)
                (retry (compute-another-Cname str))
                (begin (set! Scheme->C-names-mapping
                             (cons (cons name Cname)
                                   Scheme->C-names-mapping ) )
                       Cname ) ) ) ) ) ) )
```

Обнаруженные конфликты имён разрешаются просто: добавлением уникального номера.

```
(define (Cname-clash? Cname mapping)
  (let check ((mapping mapping))
    (and (pair? mapping)
         (or (string=? Cname (cdar mapping))
             (check (cdr mapping)) ) ) ) )

(define compute-another-Cname
  (let ((counter 1))
    (lambda (str)
      (set! counter (+ 1 counter))
      (compute-Cname (format #f "~A_~A" str counter)) ) ) )

(define (compute-Cname str)
  (define (mapcan f l)
    (if (not (pair? l)) '()
        (append (f (car l)) (mapcan f (cdr l))) ) )
  (define (convert-char char)
    (case char
      ((#\_)             '(#\_ #\_))
      ((#\?)             '(#\p))
      ((#\!)             '(#\i))
      ((#\<)             '(#\l))
      ((#\>)             '(#\g))
      ((#\=)             '(#\e))
      ((#\- #\/ #\* #\:) '())
      (else              (list char)) ) )
```

```
(let ((cname (mapcan convert-char (string->list str))))
  (if (pair? cname)
      (list->string cname)
      "weird" ) ) )
```

Единичные подчёркивания гарантированно не встречаются в генерируемых именах, поэтому их можно спокойно использовать для именования разнообразных внутренних переменных, не опасаясь коллизий с переменными исходной программы. Рассмотренное преобразование, конечно, не справится с чем-то вроде `1+`, но кого волнуют имена, запрещённые стандартом Scheme?

10.8.2. Цитаты

Цитаты должны переводиться в структуры данных Си, чтобы ими можно было пользоваться в программе. Мы поступим рационально, переводя их сразу в готовые определения структур данных, а не в код, который их создаст во время исполнения. Также мы оптимизируем занимаемый цитатами объём памяти, выделив в них и определив отдельно все общие подвыражения.

Функция `generate-quotations` назначена ответственной за выполнение данных обещаний. Для простоты мы не будем реализовывать поддержку векторов, длинной арифметики и всех нецелых чисел.

```
(define (generate-quotations out qv*)
  (when (pair? qv*)
    (format out "~%/* Quotations: */~%")
    (scan-quotations out qv* (length qv*) '()) ) )
```

Настоящую работу выполняет `scan-quotations`: просматривает собранные цитаты, представляемые объектами класса `Quotation-Variable`, и генерирует соответствующий код. Она также старается создавать как можно меньше промежуточных цитат, разделяя максимум информации между ними. Предикат `already-seen-value?` подсказывает ей, когда такое возможно.

```
(define (scan-quotations out qv* i results)
  (when (pair? qv*)
    (let* ((qv       (car qv*))
           (value    (Quotation-Variable-value qv))
           (other-qv (already-seen-value? value results)) )
      (cond
        (other-qv
          (generate-quotation-alias out qv other-qv)
          (scan-quotations out (cdr qv*) i (cons qv results)) )
        ((C-value? value)
          (generate-C-value out qv)
          (scan-quotations out (cdr qv*) i (cons qv results)) )
        ((symbol? value) (scan-symbol out value qv* i results) )
        ((pair? value)   (scan-pair out value qv* i results) )
        (else (generate-error "Unsupported constant" qv)) ) ) ) )
```

```
(define (already-seen-value? value qv*)
  (and (pair? qv*)
       (if (equal? value (Quotation-Variable-value (car qv*)))
           (car qv*)
           (already-seen-value? value (cdr qv*)) ) ) )
```

Имена всех цитат в генерируемом коде будут начинаться на `thing`. Если в программе нашлась пара эквивалентных цитат, то достаточно одну из них везде заменить другой. Функция `generate-quotation-alias` перекладывает эту работу на препроцессор Си. Чтобы облегчить чтение сгенерированного кода, рядом помещается комментарий со значением цитаты.

```
(define (generate-quotation-alias out qv1 qv2)
  (format out "#define thing~A thing~A /* ~S */~%"
          (Quotation-Variable-name qv1)
          (Quotation-Variable-name qv2)
          (Quotation-Variable-value qv2) ) )
```

Предикат `C-value?` проверяет, можно ли данную цитату прямо перевести на Си. Если да, то она передаётся функции `generate-C-value`, чтобы она выполнила этот перевод. Непосредственными значениями являются пустой список, булевы значения, (короткие) целые числа и строки. Все эти значения Scheme можно сразу представить соответствующими значениями Си. Предположим на время, что у нас уже есть необходимые макросы, раскрывающиеся в правильные описания строк, чисел, истины, лжи и пустого списка. Все они начинаются на `SCM_`. Без сомнения, гуру Си также заметят, что наши числа являются знаковыми 31-битными в дополнительном коде.

```
(define *maximal-fixnum* +1073741823)
(define *minimal-fixnum* -1073741824)

(define (C-value? value)
  (or (null? value)
      (boolean? value)
      (and (integer? value)
           (<= *minimal-fixnum* value *maximal-fixnum*) )
      (string? value) ) )

(define (generate-C-value out qv)
  (let ((value (Quotation-Variable-value qv))
        (index (Quotation-Variable-name qv)) )
    (cond
      ((null? value)
        (format out "#define thing~A SCM_nil /* () */~%"
                index ) )
      ((boolean? value)
        (format out "#define thing~A ~A /* ~S */~%"
                index (if value "SCM_true" "SCM_false") value ) )
```

```
      ((integer? value)
        (format out "#define thing~A SCM_Int2Fixnum(~A)~%"
                index value ) )
      ((string? value)
        (format out "SCM_DefineString(thing~A_object, \"~A\");~%"
                index value )
        (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
                index index ) ) ) ) )
```

Если цитируются составные значения (точечные пары или символы), то вначале они разбиваются на части, чтобы по возможности повторно использовать предыдущие определения. Символы состоят из знаков, образующих строку с их именем. Строки создаются перед символами.

```
(define (scan-symbol out value qv* i results)
  (let* ((qv (car qv*))
         (str (symbol->string value))
         (strqv (already-seen-value? str results)) )
    (cond (strqv
            (generate-symbol out qv strqv)
            (scan-quotations out (cdr qv*) i (cons qv results)) )
          (else
            (let ((newqv (make-Quotation-Variable i str)))
              (scan-quotations out (cons newqv qv*)
                               (+ i 1) results ) ) ) ) ) )

(define (generate-symbol out qv strqv)
  (format out "SCM_DefineSymbol(thing~A_object, thing~A); /* ~S */~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name strqv)
          (Quotation-Variable-value qv) )
  (format out "#define thing~A SCM_Wrap(&thing~A_object)~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name qv) ) )
```

Точечные пары состоят из двух частей — `car` и `cdr`, — обрабатываемых последовательно. При этом учитывается, что их значения могут оказаться эквивалентными как ранее определённым цитатам, так и друг другу. Их вывод реализован в стиле передачи продолжений.

```
(define (scan-pair out value qv* i results)
  (let* ((qv  (car qv*))
         (d   (cdr value))
         (dqv (already-seen-value? d results)) )
    (if dqv
        (let* ((a   (car value))
               (aqv (already-seen-value? a results)) )
          (if aqv
```

```
              (begin
                (generate-pair out qv aqv dqv)
                (scan-quotations out (cdr qv*)
                                 i (cons qv results) ) )
              (let ((newaqv (make-Quotation-Variable i a)))
                (scan-quotations out (cons newaqv qv*)
                                 (+ i 1) results ) ) ) )
        (let ((newdqv (make-Quotation-Variable i d)))
          (scan-quotations
           out (cons newdqv qv*) (+ i 1) results ) ) ) ) )

(define (generate-pair out qv aqv dqv)
  (format out
          "SCM_DefinePair(thing~A_object, thing~A, thing~A); /* ~S */~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name aqv)
          (Quotation-Variable-name dqv)
          (Quotation-Variable-value qv) )
  (format out
          "#define thing~A SCM_Wrap(&thing~A_object)~%"
          (Quotation-Variable-name qv)
          (Quotation-Variable-name qv) ) )
```

Теперь давайте рассмотрим пример, а затем поговорим обо всех этих загадочных макросах.

10.8.3. Объявление данных

Возьмём простую программу, состоящую из единственной цитаты: `(quote ((#F #T) (FOO . "FOO") 33 FOO . "FOO"))`. Её компиляция выдаёт следующий результат, слегка приправленный комментариями. Наслаждайтесь!

```
/* Compiler to C $Revision: 4.1$
'((#F #T) (FOO . "FOO") 33 FOO . "FOO") */

#include "scheme.h"

/* Quotations: */
SCM_DefineString(thing4_object, "FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing5_object, thing4);        /* FOO */
#define thing5 SCM_Wrap(&thing5_object)
SCM_DefinePair(thing3_object, thing5, thing4);  /* (FOO . "FOO") */
#define thing3 SCM_Wrap(&thing3_object)
#define thing6 SCM_Int2Fixnum(33)
SCM_DefinePair(thing2_object, thing6, thing3);  /* (33 FOO . "FOO") */
#define thing2 SCM_Wrap(&thing2_object)
```

```
SCM_DefinePair(thing1_object, thing3, thing2);
                                  /* ((FOO . "FOO") 33 FOO . "FOO") */
#define thing1 SCM_Wrap(&thing1_object)
#define thing9 SCM_nil                          /* () */
#define thing10 SCM_true                        /* #T */
SCM_DefinePair(thing8_object, thing10, thing9);  /* (#T) */
#define thing8 SCM_Wrap(&thing8_object)
#define thing11 SCM_false                       /* #F */
SCM_DefinePair(thing7_object, thing11, thing8);  /* (#F #T) */
#define thing7 SCM_Wrap(&thing7_object)
SCM_DefinePair(thing0_object, thing7, thing1);
                            /* ((#F #T) (FOO . "FOO") 33 FOO . "FOO") */
#define thing0 SCM_Wrap(&thing0_object)

/* ... */
```

Первым делом создаётся строка `"FOO"`. Для этого используется макрос `SCM_DefineString`. Первым аргументом он принимает имя нового объекта на Си. Вторым — собственно строку. Похожим образом с помощью макроса `SCM_DefineSymbol` создаётся символ. Первым аргументом идёт имя на Си, за ним — строка с именем на Scheme. Точечные пары создаются аналогично с помощью макроса `SCM_DefinePair`: первым аргументом — имя на Си, вторым и третьим — содержимое `car` и `cdr` соответственно.

Предопределённые объекты вроде булевых значений и пустого списка, конечно же, не создаются каждый раз заново. Вместо этого используются напрямую их имена: `SCM_true`, `SCM_false` и `SCM_nil`.

Позже [**см. стр. 463**] мы подробнее разберём представление в памяти значений Scheme. Процесс компиляции слабо связан с используемой конкретной реализацией структур данных. На данный момент достаточно знать лишь её интерфейс: объекты инициализируются директивами вида `SCM_Define...`, а ссылку на инициализированный объект можно получить с помощью `SCM_Wrap`. Целые числа преобразуются в объекты Scheme макросом `SCM_Int2Fixnum`. К любому объекту можно обратиться, использовав его имя, начинающееся на `thing`. Например, `thing4` означает строку `"FOO"`, а `thing6` — это число 33. Точнее, `thing4` — это указатель на объект `thing4_object`, который уже хранит настоящую строку.

Строка `"FOO"` общая для всех объектов: в выражении `(FOO . "FOO")` объекты `thing4` и `thing3` ссылаются на один и тот же `thing4_object`.

10.8.4. Компиляция выражений

Главной задачей компилятора, конечно же, является перевод выражений Scheme в выражения Си. Как было сказано ранее, мы намеренно преобразуем выражения в аналогичные выражения, а не наборы инструкций с тем же смыслом. Поступив так, мы определённо получим на выходе более понятный

код, в котором будет лучше видна структура исходной программы. Однако, несмотря на очевидную мудрость подобного решения, оно поднимает пару непростых вопросов, так как идёт вразрез с философией языка Си. В Си отдельные инструкции предпочтительнее выражений. Не то, чтобы это было особой проблемой для компилятора — ему-то без разницы; это будет проблемой для человека, который сядет отлаживать подобную программу. Беда в том, что интерактивные отладчики вроде `gdb` обычно понимают пошаговый режим исполнения как исполнение программы по инструкциям — а инструкции в нашем случае будут отнюдь не маленькие. Если бы мы решили компилировать Scheme в инструкции Си вместо выражений, то в итоге у нас получилось бы нечто похожее на рассмотренный ранее компилятор в байт-код. [**см. стр. 269**]

Компиляция выражений выполняется обобщённой функцией `->C`. Она принимает два аргумента: исходное выражение и порт, куда следует записать результат. Так как результат компиляции сразу же переносится в файл, то код необходимо генерировать строго последовательно, без возвратов и исправлений задним числом, за один проход. [**см. стр. 282**]

```
(define-generic (->C (e Program) out))
```

В отличие от предыдущих трансформаций, в этот раз мы не будем использовать обобщённый обходчик (так как у нас нет действия по умолчанию), а просто перечислим методы обработки каждого типа узлов синтаксического дерева. Эти методы довольно просты по своей структуре, так как им надо лишь правильно выводить соответствующие инструкции Си.

Язык Си имеет весьма строгий синтаксис с чётко прописанными приоритетами, ассоциативностью операций и т. д. Эти правила мало кто из программистов помнит наизусть целиком, поэтому при малейших сомнениях все пользуются скобками. Мы же возведём эту практику в абсолют и будем ставить их всегда. Что бы там ни говорили сишники о плохом вкусе, но исходным языком сейчас является Лисп, так что в программе должны быть скобочки. Много скобочек. Поэтому мы определим специальный макрос, чтобы удобно их ставить:

```
(define-syntax between-parentheses
  (syntax-rules ()
    ((between-parentheses out . body)
     (let ((out out))
       (format out "(")
       (begin . body)
       (format out ")") ) ) ) )
```

Обращения к переменным

В программе могут использоваться переменные разного вида, но в общем случае одна переменная Scheme соответствует одной переменной Си. Метод

обработки обращений к переменным сразу же перепоручает всю работу обобщённой функции `reference->C`, а она, в свою очередь, по умолчанию просит всё сделать следующую функцию — `variable->C`. Такая слабая связанность методов облегчит их дальнейшую специализацию.

```
(define-method (->C (e Reference) out)
  (reference->C (Reference-variable e) out) )

(define-generic (reference->C (v Variable) out)
  (variable->C v out) )

(define-generic (variable->C (variable) out))
```

Обычно переменная получает новое имя на Си, похожее на её исходное, за исключением случаев, когда она была переименована или же ссылается на цитату.

```
(define-method (variable->C (variable Variable) out)
  (format out (IdScheme->IdC (Variable-name variable))) )

(define-method (variable->C (variable Renamed-Local-Variable) out)
  (format out "~A_~A"
          (IdScheme->IdC (Variable-name variable))
          (Renamed-Local-Variable-index variable) ) )

(define-method (variable->C (variable Quotation-Variable) out)
  (format out "thing~A" (Quotation-Variable-name variable)) )
```

Отдельно обрабатываются обращения к глобальным переменным, не являющимся предопределёнными. Фактически, это свободные переменные всей программы. Как вы помните, в данной реализации подобные переменные создаются автоматически, при их первом упоминании. К сожалению, компилятор никак не анализирует значения этих переменных, поэтому проверки на инициализированность должны проводиться во время исполнения программы. За этим проследит макрос `SCM_CheckedGlobal`. [**см. упр. 10.2**]

```
(define-method (reference->C (v Global-Variable) out)
  (format out "SCM_CheckedGlobal")
  (between-parentheses out
    (variable->C v out) ) )
```

Кроме того, ещё остаются обычные свободные переменные. Для них тоже припасён специальный макрос.

```
(define-method (->C (e Free-Reference) out)
  (format out "SCM_Free")
  (between-parentheses out
    (variable->C (Free-Reference-variable e) out) ) )
```

Присваивания

С присваиваниями всё ещё легче, так как они бывают лишь двух видов: присваивание глобальным переменным и модификация содержимого коробок. Случай глобальных переменных прямо переводится в присваивание соответствующей глобальной переменной языка Си.

```
(define-method (->C (e Global-Assignment) out)
  (between-parentheses out
    (variable->C (Global-Assignment-variable e) out)
    (format out " = ")
    (->C (Global-Assignment-form e) out) ) )
```

Коробки

Что касается коробок, то над ними возможны всего три операции. Макрос `SCM_Content` позволяет смотреть или изменять содержимое коробок, а библиотечная функция `SCM_allocate_box` берёт на себя их создание.

```
(define-method (->C (e Box-Read) out)
  (format out "SCM_Content")
  (between-parentheses out
    (->C (Box-Read-reference e) out) ) )

(define-method (->C (e Box-Write) out)
  (between-parentheses out
    (format out "SCM_Content")
    (between-parentheses out
      (->C (Box-Write-reference e) out) )
    (format out " = ")
    (->C (Box-Write-form e) out) ) )

(define-method (->C (e Box-Creation) out)
  (variable->C (Box-Creation-variable e) out)
  (format out " = SCM_allocate_box")
  (between-parentheses out
    (variable->C (Box-Creation-variable ) out) ) )
```

Ветвления

К счастью, в Си есть тернарный оператор, позволяющий записывать ветвления выражениями вида $(\pi_0 \ ? \ \pi_1 \ : \ \pi_2)$. Так как в Scheme любое значение, не тождественное `#f`, считается истиной, мы должны это явно проверять. И, конечно, не забываем о скобках!

```
(define-method (->C (e Alternative) out)
  (between-parentheses out
```

```
    (boolean->C (Alternative-condition e) out)
    (format out "~%? ")
    (->C (Alternative-consequent e) out)
    (format out "~%: ")
    (->C (Alternative-alternant e) out) ) )

(define-generic (boolean->C (e) out)
  (between-parentheses out
    (->C e out)
    (format out " != SCM_false") ) )
```

Внимательный читатель уже заметил, что проверка условия не всегда выполняется с максимальной эффективностью. В программах очень часто встречаются выражения вида `(if (pair? x) ...)`. Вызов предиката `pair?` здесь честно возвращает логические значения Scheme, но в данном случае он мог бы сразу выдать значение, удобное для Си. Мелочь, конечно, но копейка рубль бережёт. Одним вариантом оптимизации является специализация функции `boolean->C` для вызовов предопределённых предикатов. Также можно отождествить логические значения Scheme и Си, условившись считать ложью значение `NULL`. Наконец, частичный вывод типов выражений позволяет избавиться от этой и прочих излишних проверок и преобразований, см. [Shi91, Ser93, WC94].

Последовательные вычисления

Для последовательного вычисления подвыражений в Си тоже есть специальная конструкция: `(`π_1`, ..., `π_n`)`.

```
(define-method (->C (e Sequence) out)
  (between-parentheses out
    (->C (Sequence-first e) out)
    (format out ",~%")
    (->C (Sequence-last e) out) ) )
```

10.8.5. Компиляция аппликаций

Мы разделили аппликации функций на несколько категорий: нормальные вызовы функций, приводимые формы (аппликации, где на месте функции находится абстракция) и обращения к предопределённым примитивам. Этим трём категориям аппликаций соответствуют узлы синтаксического дерева `Regular-Application`, `Fix-Let` и `Predefined-Application`.

Нормальные вызовы функций вида `(`$f\ x_1\ \ldots\ x_n$`)` преобразуются в выражения Си `SCM_invoke(`f`, `n`, `x_1`, ..., `x_n`)`, где n — это количество аргументов, переданных функции, а `SCM_invoke` — специальная вспомогательная функция. Мы определим несколько макросов, чтобы сделать вызовы функций с малым числом аргументов более читабельными:

```
#define SCM_invoke0(f)          SCM_invoke(f, 0)
#define SCM_invoke1(f, x)       SCM_invoke(f, 1, x)
#define SCM_invoke2(f, x, y)    SCM_invoke(f, 2, x, y)
#define SCM_invoke3(f, x, y, z) SCM_invoke(f, 3, x, y, z)
```

Вызовы большей арности компилируются напрямую в `SCM_invoke`:

```
(define-method (->C (e Regular-Application) out)
  (let ((n (number-of (Regular-Application-arguments e))))
    (cond ((< n 4)
           (format out "SCM_invoke~A" n)
           (between-parentheses out
             (->C (Regular-Application-function e) out)
             (->C (Regular-Application-arguments e) out) ) )
          (else
           (format out "SCM_invoke")
           (between-parentheses out
             (->C (Regular-Application-function e) out)
             (format out ", ~A" n)
             (->C (Regular-Application-arguments e) out) ) ) ) ) )
```

Приводимые формы просто переводятся в последовательность выражений Си, соответствующих их телу. Наконец-то усилия по предварительному анализу кода и идентификации временных переменных начинают приносить плоды. Благодаря им становится возможным выполнить так называемое *объединение фреймов* (frame coalescing); это известная оптимизация [AS94, PJ87], касающаяся временных переменных: вместо множества маленьких выделений-освобождений памяти сразу взять один большой кусок для целого набора переменных. А если под все временные переменные уже выделена память, то связывание их со значениями сводится лишь к присваиваниям. В терминах Лиспа данное преобразование приводит к объединению локальных `let`-форм, переносу их на уровень выше и возможному переименованию соответствующих переменных.[4]

```
(begin                          (let (x1 x2)
  e1                              e1
  (let ((x π1))                   (set! x1 π1)
    π2 )              ⇝           π2[x → x1]
  e2                              e2
  (let ((x π3))                   (set! x2 π3)
    π4 ) )                        π4[x → x2] )
```

В общем случае это преобразование ошибочно: например, если благодаря продолжениям e_1 вернёт значение во второй раз, а π_2 захватывает значение `x`,

[4] Конечно, при этом можно применять и более агрессивные оптимизации. Например, в приведённом примере переменные `x1` и `x2` невозможно использовать одновременно, так что они могут физически совпадать в памяти.

то новая переменная x, переименованная в x1, сохранит привязку x из параллельной ветви стека. [**см. стр. 229**] Тем не менее, в данном случае всё хорошо, так как клоны x1 всё же будут отличаться: значение x1 будет завёрнуто в коробку, а вместо set! будет Box-Write; так что хоть коробка и останется той же, но привязка для её содержимого будет создана заново.

Остаётся только правильно инициализировать каждую локальную переменную. Так как ранее был проведён их анализ и переименование, то конфликтов областей видимости не будет. Простая обобщённая функция bindings->C без проблем справится с инициализацией.

```
(define-method (->C (e Fix-Let) out)
  (between-parentheses out
    (bindings->C (Fix-Let-variables e) (Fix-Let-arguments e) out)
    (->C (Fix-Let-body e) out) ) )

(define-generic (bindings->C variables (arguments) out))

(define-method (bindings->C variables (e Arguments) out)
  (variable->C (car variables) out)
  (format out " = ")
  (->C (Arguments-first e) out)
  (format out ",~%")
  (bindings->C (cdr variables) (Arguments-others e) out) )

(define-method (bindings->C variables (e No-Argument) out)
  (format out "") )
```

Наконец, остаётся случай аппликации, когда функцией является значение предопределённой переменной, — такие вызовы *встраиваются* напрямую в код. Примитивам нет нужды пользоваться SCM_invoke. Они будут написаны на Си и включены в состав библиотеки времени исполнения, которая подключается к любой скомпилированной программе. Примитивы знают о внутренних структурах данных, так что могут обойтись без относительно дорогого (и бесполезного в данном случае) вызова SCM_invoke.

Так как точная форма прямого вызова примитива зависит от его реализации, мы поступим следующим образом: расширим класс дескрипторов, описывающих функции-примитивы, дополнительным полем generator, служащим для хранения специальных функций-генераторов. Их задачей будет принять интересующую нас аппликацию и выдать в порт соответствующий код на Си. Каким образом они его получат — это личное дело генераторов, но чтобы облегчить им немного жизнь, мы определим обобщённую функцию arguments->C, рекурсивно применяющую ->C к аргументам вызова.

```
(define-generic (arguments->C (e) out))

(define-method (arguments->C (e No-Argument) out) #t)
```

```
(define-method (arguments->C (e Arguments) out)
  (->C (Arguments-first e) out)
  (->C (Arguments-others e) out) )

(define-method (->C (e Predefined-Application) out)
  ((Functional-Description-generator
    (Predefined-Variable-description
     (Predefined-Application-variable e) ) ) e out ) )
```

10.8.6. Предопределённое окружение

Если мы собрались пользоваться предопределёнными функциями, то о них вначале необходимо рассказать компилятору. Для этого определяется специальный макрос `defprimitive`:

```
(define-class Functional-Description Object
  ( comparator arity generator ) )

(define-syntax defprimitive
  (syntax-rules ()
    ((defprimitive name Cname arity)
     (let ((v (make-Predefined-Variable 'name
               (make-Functional-Description
                = arity
                (make-predefined-application-generator 'Cname) ) )))
       (set! g.init (cons v g.init))
       'name ) ) ) )
```

Итак, компилятору известно имя примитива на Scheme и Си, а также его арность. На выходе ожидается вызов функции Си: то есть имя функции, за которым в скобках следуют аргументы, разделённые запятыми. Функция `make-predefined-application-generator` создаёт генераторы подобных вызовов.

```
(define (make-predefined-application-generator Cname)
  (lambda (e out)
    (format out "~A" Cname)
    (between-parentheses out
      (arguments->C (Predefined-Application-arguments e) out) ) ) )
```

Приведём несколько примеров для всем знакомых функций:

```
(defprimitive cons "SCM_cons" 2)
(defprimitive car "SCM_car" 1)
(defprimitive + "SCM_Plus" 2)
(defprimitive = "SCM_EqnP" 2)
```

Отсюда видно, что вызов `cons` будет компилироваться в вызов функции `SCM_cons`, а примитив `+` — в макрос `SCM_Plus`. [**см. стр.** 468] Макрос можно

отличить от функции по прописной первой букве имени. Некоторые примитивы из соображений компактности кода эффективнее реализовывать именно как макросы.

10.8.7. Компиляция функций

Выполненные ранее трансформации преобразовали все абстракции в явно создаваемые замыкания, которым соответствуют объекты класса `Closure-Creation`. В Си за создание замыканий будет отвечать библиотечная функция `SCM_close`. Первым аргументом она принимает адрес Си-функции (завёрнутый в макрос `SCM_CfunctionAddress`), соответствующий телу абстракции; за адресом следует арность создаваемого замыкания, а в конце списка расположены количество замыкаемых значений и, собственно, сами значения. Данные значения извлекаются из поля `free` обрабатываемой абстракции. Количество аргументов функций с фиксированной арностью записывается просто соответствующим числом. Но если это функция с переменной арностью, то ей может быть передано любое количество аргументов, не меньшее некоторого i; этот случай будет представляться числом $-i-1$. К примеру, арность функции `list` равна -1.

```
(define-method (->C (e Closure-Creation) out)
  (format out "SCM_close")
  (between-parentheses out
    (format out "SCM_CfunctionAddress(function_~A), ~A, ~A"
            (Closure-Creation-index e)
            (generate-arity (Closure-Creation-variables e))
            (number-of (Closure-Creation-free e)) )
    (->C (Closure-Creation-free e) out) ) )

(define (generate-arity variables)
  (let count ((variables variables) (arity 0))
    (if (pair? variables)
        (if (Local-Variable-dotted? (car variables))
            (- (+ arity 1))
            (count (cdr variables) (+ 1 arity)) )
        arity ) ) )

(define-method (->C (e No-Free) out)
  #t )

(define-method (->C (e Free-Environment) out)
  (format out ", ")
  (->C (Free-Environment-first e) out)
  (->C (Free-Environment-others e) out) )
```

Все абстракции компилируемой программы собираются в соответствующем поле объекта `Flattened-Program` в виде списка объектов класса `With-`

`Temp-Function-Definition`. Каждому из них сопоставляется эквивалентная функция на Си.

```
(define (generate-functions out definitions)
  (format out "~%/* Functions: */~%")
  (for-each (lambda (def)
              (generate-closure-structure out def)
              (generate-possibly-dotted-definition out def) )
            (reverse definitions) ) )
```

Для повышения удобочитаемости генерируемого кода снова используются макросы, чтобы скрыть детали реализации. Для каждого замыкания генерируется функция, содержащая его тело, а также структура данных, которая будет хранить замкнутые переменные. Имена создаваемых сущностей строятся по простому принципу: корень `function_` + номерной суффикс, взятый из поля `index` объекта `Function-Definition`.[5]

Внутренняя структура замыканий формируется макросом `SCM_DefineClosure`. Первым аргументом он получает номер соответствующей функции, а вторым — имена захватываемых замыканием переменных, разделённые точками с запятой.

```
(define (generate-closure-structure out definition)
  (format out "SCM_DefineClosure(function_~A, "
          (Function-Definition-index definition) )
  (generate-local-temporaries (Function-Definition-free definition)
                              out )
  (format out ");~%") )
```

Функция `generate-possibly-dotted-definition` генерирует определение тела замыкания в виде функции Си, учитывая его арность. Прототип этой функции формирует макрос `SCM_DeclareFunction`, принимающий её имя как аргумент. Локальные переменные функции определяются с помощью макросов `SCM_DeclareLocalVariable` и `SCM_DeclareLocalDottedVariable`. Они принимают имя создаваемой переменной, а также её порядковый номер в списке переменных функции. Правда, этот номер нужен только для сборки списка, связываемого с точечной переменной. С телом функции вообще нет проблем: просто применяем к нему `->C` и не забываем вернуть вычисленное значение с помощью оператора `return`.

```
(define (generate-possibly-dotted-definition out definition)
  (format out "~%SCM_DeclareFunction(function_~A)~%{~%"
          (Function-Definition-index definition) )
  (let ((vars (Function-Definition-variables definition))
        (index -1) )
    (for-each (lambda (v)
```

[5] Имена функций вида `function_`i, очевидно, никак не связаны с именами соответствующих им глобальных переменных. Если не пренебрегать таблицей символов, как это сделано здесь, то вполне можно выводить более понятные имена.

```
                (set! index (+ 1 index))
                (cond ((Local-Variable-dotted? v)
                       (format out "SCM_DeclareLocalDottedVariable(") )
                      ((Variable? v)
                       (format out "SCM_DeclareLocalVariable(") ) )
                (variable->C v out)
                (format out ", ~A);~%" index) )
              vars )
    (let ((temps (With-Temp-Function-Definition-temporaries
                  definition )))
      (when (pair? temps)
        (generate-local-temporaries temps out)
        (format out "~%") ) )
    (format out "return ")
    (->C (Function-Definition-body definition) out)
    (format out ";~%}~%~%") ) )
```

Список локальных переменных формируется функцией `generate-local-temporaries`:

```
(define (generate-local-temporaries temps out)
  (when (pair? temps)
    (format out "SCM ")
    (variable->C (car temps) out)
    (format out "; ")
    (generate-local-temporaries (cdr temps) out) ) )
```

10.8.8. Последний штрих

Сейчас наш компилятор способен переварить любую программу на Scheme, но для получения полноценной программы на Си ей не хватает функции `main`. Именно этой функции соответствует то замыкание, в которое была обернута вся программа, когда мы избавлялись от вложенных `lambda`-форм. [**см. стр. 437**] Единственным спорным вопросом здесь является включение в главную функцию вызова `SCM_print`, выводящего на экран результат работы программы. В общем случае это не нужно — есть множество полезных и при этом молчаливых программ (например, `gcc`). Но для нас более удобным будет сразу же видеть достижения наших небольших программок.

```
(define (generate-main out form)
  (format out "~%/* Expression: */~%")
  (format out "int main(void)~%{~%")
  (format out "    SCM_print")
  (between-parentheses out
    (->C form out) )
  (format out ";~%    return 0;~%}~%") )
```

Конечно, о представлении данных ещё не было сказано ничего конкретного, равно как и о реализации библиотеки времени исполнения, но у нас уже есть полностью рабочий компилятор вместе с огромным желанием испытать его в деле. Натравим его на подопытный пример, приведённый в начале главы. [**см. стр. 428**] И-и-и… вот что получается на выходе (отступы расставлены вручную для читабельности):

o/chap10ex.c

```
/* Compiler to C $Revision: 4.1$
(BEGIN
  (SET! INDEX 1)
  ((LAMBDA
     (CNTER . TMP)
     (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
     (IF CNTER (CNTER TMP) INDEX))
    (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
    'FOO))
*/

#include "scheme.h"

/* Global environment: */
SCM_DefineGlobalVariable(INDEX, "INDEX");

/* Quotations: */
#define thing3 SCM_nil                          /* () */
SCM_DefineString(thing4_object, "FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing2_object, thing4);        /* FOO */
#define thing2 SCM_Wrap(&thing2_object)
#define thing1 SCM_Int2Fixnum(1)
#define thing0 thing1                           /* 1 */

/* Functions: */
SCM_DefineClosure(function_0, );

SCM_DeclareFunction(function_0)
{
    SCM_DeclareLocalVariable(F, 0);
    return ((INDEX = SCM_Plus(thing1,
                              SCM_CheckedGlobal(INDEX))),
            SCM_invoke1(F,
                        SCM_CheckedGlobal(INDEX)));
}

SCM_DefineClosure(function_1, SCM I; );
```

```
SCM_DeclareFunction(function_1)
{
    SCM_DeclareLocalDottedVariable(X, 0);
    return SCM_cons(SCM_Free(I),
                    X);
}

SCM_DefineClosure(function_2, );

SCM_DeclareFunction(function_2)
{
    SCM_DeclareLocalVariable(I, 0);
    return SCM_close(SCM_CfunctionAddress(function_1), -1, 1, I);
}

SCM_DefineClosure(function_3, );

SCM_DeclareFunction(function_3)
{
    SCM TMP_2; SCM CNTER_1;
    return ((INDEX = thing0),
            (CNTER_1 = SCM_close(SCM_CfunctionAddress(function_0), 1, 0),
             TMP_2 = SCM_cons(thing2,
                              thing3),
             (TMP_2 = SCM_allocate_box(TMP_2),
              ((SCM_Content(TMP_2) =
               SCM_invoke1(CNTER_1, SCM_close(SCM_CfunctionAddress
                                              (function_2), 1, 0))),
               ((CNTER_1 != SCM_false)
                ? SCM_invoke1(CNTER_1,
                              SCM_Content(TMP_2))
                : SCM_CheckedGlobal(INDEX))))));
}

/* Expression: */
int main(void)
{
    SCM_print(SCM_invoke0(SCM_close(SCM_CfunctionAddress(function_3),
                                    0, 0)));
    return 0;
}

/* End of generated code */
```

Также стоит взглянуть на функции `function_1` и `function_2` в раскрытом виде, без макросов (кроме `va_arg`). Остальные выражения раскрываются аналогично.

```
struct function_1 {
    SCM     (*behavior)(void);
    long    arity;
    SCM     I;
};

SCM function_1(struct function_1 *self_,
               unsigned long size_,
               va_list arguments_)
{
    SCM X = SCM_list(size_ - 0, arguments_);
    return SCM_cons(((self_)->I), X);
}

struct function_2 {
    SCM     (*behavior)(void);
    long    arity;
};

SCM function_2(struct function_2 *self_,
               unsigned long size_,
               va_list arguments_)
{
    SCM I = va_arg(arguments_, SCM);
    return SCM_close(((SCM(*)(void)) function_1), -1, 1, I);
}
```

Если откомпилировать получившуюся программу любым компилятором, поддерживающим стандарт C90 (например, `gcc`), подключив к ней необходимые библиотеки, о которых мы упоминали, то получится (весьма небольшой) исполнимый файл, который (весьма быстро[6]) выдаёт ожидаемый результат.[7]

```
% gcc -ansi -pedantic chap10ex.c scheme.o schemelib.o
% time a.out
(2 3)
0.010u 0.000s 0:00.00 0.0% 3+5k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
28762   4096    32      32800   8020
```

[6] Большая часть потраченного времени уходит на загрузку; собственно исполнение занимает пренебрежимо малую его часть. Если данные вычисления повторить 10 000 раз, то время выполнения программы увеличится всего до одной секунды.

[7] Замеры производились на Sony News 3200 с процессором MIPS R3000 на борту.

10.9. Представление данных

В этом разделе мы займёмся разбором макросов, описываемых в файле `"scheme.h"`. Нет нужды повторять, что целью данного *упражнения* является ознакомление, объяснение и демонстрация разнообразных приёмов, а не создание самого быстрого и эффективного в мире компилятора. Поэтому мы не будем обременять себя проблемами управления памятью; все выделения памяти выполняются простыми вызовами `malloc`, так что при желании будет весьма просто подключить к программам сторонний консервативный сборщик мусора, например, известный Boehm GC [BW88].

В Лиспе и Scheme принята динамическая типизация, то есть тип любого значения всегда можно узнать во время исполнения программы. Соответственно, эту информацию необходимо где-то хранить. Одной из основных головных болей разработчиков реализаций динамически типизированных языков является как раз эффективное связывание значений с их типами. Мало того, значения ещё могут быть различного размера, так как некоторые из них представляют собой произвольные наборы других значений. С этой проблемой можно справиться, применив косвенную адресацию — указатели всегда имеют постоянный размер. Объекты будут динамически создаваться в памяти, обращаться к ним будем посредством указателей, а их типы можно размещать, например, в начале выделенных фрагментов памяти. Неудобство (по сравнению со статически типизированными языками, где вопросы с типами уже решены до исполнения программы) здесь в том, что простые значения вроде чисел не могут быть эффективно использованы напрямую, только через указатели. Ситуацию можно исправить несколькими способами. Один из них основан на том, что в современных компьютерах данные в памяти выравниваются по границе четырёх или восьми байт, что оставляет два-три младших бита адресов неиспользуемыми. Этими битами можно закодировать информацию о виде значения, на которое ссылается указатель. А если значение является числом, то почему бы не записать его вместо адреса? Мы так и поступим с целыми числами, сделав работу с ними более эффективной, но отдав за это один значащий бит, что несколько уменьшит диапазон непосредственно представимых значений. Конечно, есть и другие варианты, неплохую их подборку можно найти в [Gud93].

Возвращаемся к насущному; взгляните на рисунок 10.5. Значения Scheme представляются такими значениями Си, что если их младший бит равен единице, то остальные биты считаются представлением некоторого целого числа. Например, на 32-битных машинах получаются 31-битные числа. Если же младший бит нулевой, то это адрес первого поля соответствующего значения Scheme. Тип этого значения расположен *перед*[8] его полями, как у [DPS94a].

[8] Пустой указатель в Си (`NULL`) чаще всего численно равен нулю. Естественно, он не является допустимым значением Scheme, так как нелегко будет отыскать в памяти адрес -1.

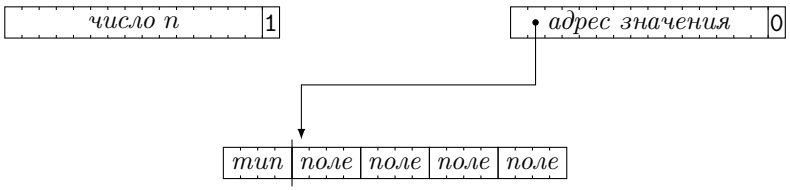


Рис. 10.5. Представление значений Scheme в Си.

Сами значения представляются обычными структурами данных Си. Их тип — как вы уже, наверное, догадались — называется `SCM`.

```
typedef union SCM_object *SCM;
```

Макрос `SCM_FixnumP` позволяет отличить целые числа от указателей. Приведением типов занимаются макросы `SCM_Fixnum2Int` и `SCM_Int2Fixnum`. Например, числу 37 в Scheme соответствует число 75 в Си.

```
#define SCM_FixnumP(x)    ((unsigned long)(x) & 1UL)
#define SCM_Fixnum2Int(x) ((long)(x) >> 1)
#define SCM_Int2Fixnum(i) ((SCM)(((i) << 1) | 1UL))
```

Если значение `SCM` не является числом, то это указатель на объект типа `SCM_object`. Данное объединение все типы данных, поддерживаемые языком. Некоторых типов там нет, в частности: чисел с плавающей запятой, векторов, портов. Но самое важное в Scheme — точечные пары,[9] замыкания и продолжения — мы обязательно реализуем.

```
union SCM_object {
    struct SCM_pair {
        SCM cdr;
        SCM car;
    } pair;

    struct SCM_string {
        char Cstring[8];
    } string;

    struct SCM_symbol {
        SCM pname;
    } symbol;

    struct SCM_box {
        SCM content;
    } box;
```

[9] Как вы помните, `cdr` лучше располагать перед `car`, потому что пары — это всё же основной элемент связных списков, а `cdr` при обходе списка просматривается чаще, как утверждается в [Cla79].

```
        struct SCM_subr {
            SCM (*behavior)(void);
            long arity;
        } subr;

        struct SCM_closure {
            SCM (*behavior)(void);
            long arity;
            SCM environment[1];
        } closure;

        struct SCM_escape {
            struct SCM_jmp_buf *stack_address;
        } escape;
    };
```

Полноценные значения Scheme, представляемые `SCM_object`, должны содержать внутри себя собственный тип. Мы будем его хранить весьма специфичным образом. Собственно метка типа представляется перечислением `SCM_tag`. Числовые константы выбраны так, чтобы гарантировать некоторое количество незначащих битов, например, для нужд сборщика мусора. В памяти метка должна располагаться не абы как, а быть строго выровненной по размеру `SCM` — именно для этого в объединение `SCM_header` добавлена заглушка `ignored`. Наконец, объекты вместе с информацией об их типе будут представляться объединениями `SCM_unwrapped_object`.

```
    enum SCM_tag {
        SCM_NULL_TAG      = 0xAAA0,
        SCM_PAIR_TAG      = 0xAAA1,
        SCM_BOOLEAN_TAG   = 0xAAA2,
        SCM_UNDEFINED_TAG = 0xAAA3,
        SCM_SYMBOL_TAG    = 0xAAA4,
        SCM_STRING_TAG    = 0xAAA5,
        SCM_SUBR_TAG      = 0xAAA6,
        SCM_CLOSURE_TAG   = 0xAAA7,
        SCM_ESCAPE_TAG    = 0xAAA8,
    };

    union SCM_header {
        enum SCM_tag tag;
        SCM ignored;
    };

    union SCM_unwrapped_object {
        struct SCM_unwrapped_immediate_object {
            union SCM_header header;
        };
```

```
    struct SCM_unwrapped_pair {
        union SCM_header header;
        SCM cdr;
        SCM car;
    } pair;

    struct SCM_unwrapped_string {
        union SCM_header header;
        char Cstring[8];
    } string;

    struct SCM_unwrapped_symbol {
        union SCM_header header;
        SCM pname;
    } symbol;

    struct SCM_unwrapped_subr {
        union SCM_header header;
        SCM (*behavior)(void);
        long arity;
    } subr;

    struct SCM_unwrapped_closure {
        union SCM_header header;
        SCM (*behavior)(void);
        long arity;
        SCM environment[1];
    } closure;

    struct SCM_unwrapped_escape {
        union SCM_header header;
        struct SCM_jmp_buf *stack_address;
    } escape;
};
```

Следующие макросы помогают с приведением и определением типов объектов. Они будут полезными по большей части лишь библиотечным функциями, потому как только им необходимо различать `SCM` и `SCMref`.

```
typedef union SCM_unwrapped_object *SCMref;

#define SCM_Wrap(x)   ((SCM)    (((union SCM_header*) x) + 1))
#define SCM_Unwrap(x) ((SCMref) (((union SCM_header*) x) - 1))
#define SCM_2tag(x)   ((SCM_Unwrap((SCM) x))->object.header.tag)
```

Наконец, адреса функций, написанных на Си, будут храниться (но не использоваться) как указатели на функции без аргументов, возвращающие SCM. Макрос SCM_CfunctionAddress приводит их к этому типу.

```
#define SCM_CfunctionAddress(Cfunction) ((SCM (*)(void)) Cfunction)
```

10.9.1. Объявления значений

Кроме всего прочего, scheme.h определяет специальные макросы для статического создания значений Scheme. Их имена начинаются на SCM_Define. Чтобы получить точечную пару, достаточно объявить новую переменную типа SCM_unwrapped_pair. Символы и прочие значения объявляются аналогично. Всякий раз сначала создаётся объект, а затем берётся его адрес и превращается в корректное значение Scheme с помощью SCM_Wrap.

```
#define SCM_DefinePair(pair, car, cdr)  \
  static struct SCM_unwrapped_pair pair = {{SCM_PAIR_TAG}, cdr, car}
#define SCM_DefineSymbol(symbol, name)  \
  static struct SCM_unwrapped_symbol symbol = {{SCM_SYMBOL_TAG}, name}
```

Со строками всё немного сложнее, так как в Си нельзя статически инициализировать структуры переменного размера. Также следует быть очень осторожными со строками Си: не перепутать их содержимое с адресом и не забыть о нулевом байте в конце. Мы решим данные затруднения, объявляя для каждой статической строки личную структуру. Собственно содержимое строк будет представляться родными строками Си.

```
#define SCM_DefineString(Cname, string) \
  struct Cname##_struct {               \
      union SCM_header header;          \
      char Cstring[1 + sizeof(string)]; \
  };                                    \
  static struct Cname##_struct Cname =  \
      {{SCM_STRING_TAG}, string}
```

Также нам будут необходимы несколько предопределённых значений. Для программ важны только их имена и типы, поэтому они объявляются как непосредственные значения. Мы воспользуемся SCM_Wrap для приведения их к типу SCM.

```
#define SCM_DefineImmediateObject(name, tag) \
  struct SCM_unwrapped_immediate_object name = {{tag}}

SCM_DefineImmediateObject(SCM_true_object,  SCM_BOOLEAN_TAG)
SCM_DefineImmediateObject(SCM_false_object, SCM_BOOLEAN_TAG)
SCM_DefineImmediateObject(SCM_nil_object,   SCM_PAIR_TAG)

#define SCM_true  SCM_Wrap(&SCM_true_object)
#define SCM_false SCM_Wrap(&SCM_false_object)
```

```
#define SCM_nil   SCM_Wrap(&SCM_nil_object)
```

Логические значения Scheme не тождественны логическим значениям Си, поэтому соответствующее преобразование не будет лишним. Определим для этого макрос `SCM_2bool`:

```
#define SCM_2bool(i) ((i) ? SCM_true : SCM_false)
```

Также мы определим несколько вспомогательных макросов для проверки типов и простейших операций со значениями. Предикаты имеют суффикс `P`, который означает, что они возвращают логические значения Си.

```
#define SCM_Car(x)     (SCM_Unwrap(x)->pair.car)
#define SCM_Cdr(x)     (SCM_Unwrap(x)->pair.cdr)
#define SCM_NullP(x)   ((x) == SCM_nil)
#define SCM_EqP(x, y)  ((x) == (y))

#define SCM_PairP(x)   \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x) == SCM_PAIR_TAG))

#define SCM_SymbolP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x) == SCM_SYMBOL_TAG))

#define SCM_StringP(x) \
  ((!SCM_FixnumP(x)) && (SCM_2tag(x) == SCM_STRING_TAG))
```

Естественно, макросы вроде `SCM_Car` и `SCM_Cdr` будут использоваться исключительно в тех случаях, когда компилятор *знает*, что значение имеет подходящий тип. Следующие макросы для арифметических действий, например, этого заранее знать не могут, поэтому в них проверки типов выполняются явно. Конечно, таким образом мы лишаемся возможной оптимизации в случае, если типы всё же будут известны,[10] но этот вопрос уже обсуждался: с точки зрения педагогики важнее показать всю массу вариантов, нежели рассмотреть в деталях пару-тройку лучших. Если бы мы хотели добиться максимальной эффективности, компилятор был бы устроен совсем не так и выдавал бы совершенно иной код.

```
#define SCM_Plus(x, y)                                    \
  ((SCM_FixnumP(x) && SCM_FixnumP(y))                     \
   ? SCM_Int2Fixnum(SCM_Fixnum2Int(x) + SCM_Fixnum2Int(y)) \
   : SCM_error(SCM_ERR_PLUS))

#define SCM_GtP(x, y)                                     \
  ((SCM_FixnumP(x) && SCM_FixnumP(y))                     \
   ? SCM_2bool(SCM_Fixnum2Int(x) > SCM_Fixnum2Int(y))     \
   : SCM_error(SCM_ERR_GTP))
```

[10] Некоторые компиляторы Си *достаточно сообразительны*, чтобы избавиться от части ненужных проверок, но, как известно, хочешь сделать хорошо — сделай это сам.

10.9.2. Глобальные переменные

Для получения значения глобальной переменной используется макрос SCM_CheckedGlobal, который проверяет, была ли она инициализирована или нет. Следовательно, необходимо выделить какое-то определённое значение Си для обозначения неинициализированных переменных Scheme — назовём его SCM_undefined. Изменяемые глобальные переменные при создании просто инициализируются (в Си) этим значением, показывающим, что (в Scheme) они ещё не были инициализированы.

```
SCM_DefineImmediateObject(SCM_undefined_object, SCM_UNDEFINED_TAG);
#define SCM_undefined SCM_Wrap(&SCM_undefined_object)

#define SCM_CheckedGlobal(Cname) \
  ((Cname != SCM_undefined) ? Cname
                            : SCM_error(SCM_ERR_UNINITIALIZED))

#define SCM_DefineInitializedGlobalVariable(Cname, string, value) \
  SCM Cname = SCM_Wrap(value)

#define SCM_DefineGlobalVariable(Cname, string)      \
  SCM_DefineInitializedGlobalVariable(Cname, string, \
                                      &SCM_undefined_object)
```

Ещё одной важным делом является связывание предопределённых значений с предопределёнными переменными. Например, если мы хотим пользоваться в программах NIL для обозначения пустого списка (), то следует определить в Си глобальную переменную NIL со значением SCM_nil_object. Это определение (вместе с множеством других) находится в файле schemelib.c:

```
SCM_DefineInitializedGlobalVariable(NIL, "NIL", &SCM_nil_object);
SCM_DefineInitializedGlobalVariable(F,   "F",   &SCM_false_object);
SCM_DefineInitializedGlobalVariable(T,   "T",   &SCM_true_object);
```

Многое можно выразить с помощью этих трёх значений, но нам бы хотелось иметь в распоряжении также и знакомые функции: CONS, CAR и т. д. Они определяются макросом SCM_DefinePredefinedFunctionVariable. Вот его определение вместе с несколькими примерами:

```
#define SCM_DefinePredefinedFunctionVariable(subr, string,          \
                                             arity, Cfunction)      \
  static struct SCM_unwrapped_subr subr##_object =                  \
      {{SCM_SUBR_TAG}, Cfunction, arity};                           \
  SCM_DefineInitializedGlobalVariable(subr, string, &(subr##_object))

SCM_DefinePredefinedFunctionVariable(CAR,  "CAR",  1, SCM_car);
SCM_DefinePredefinedFunctionVariable(CONS, "CONS", 2, SCM_cons);
SCM_DefinePredefinedFunctionVariable(EQN,  "=",    2, SCM_eqnp);
SCM_DefinePredefinedFunctionVariable(EQ,   "EQ?",  2, SCM_eqp);
```

Значения изменяемых глобальных переменных размещаются в коробках. Просмотр или изменение подобных значений выполняются с помощью макроса `SCM_Content`, определяемого следующим образом:

```
#define SCM_Content(e) ((e)->box.content)
```

10.9.3. Определения функций

Осталось лишь рассмотреть представление в памяти функций. Важно тщательно проработать этот аспект, так как от него существенно зависят возможности взаимодействия Scheme и Си.

Примитивы с фиксированной арностью представляются функциями Си той же арности. Следовательно, функцию `cons` можно будет вызвать простым и легко запоминающимся способом: `SCM_cons(`a`, `d`)`.

Остальные функции, являющиеся полноценными объектами Scheme, требуют бóльших усилий. Им может быть передано любое количество аргументов. Они могут быть результатами вычислений. При вызовах через `apply` аргументы им передаются по-особому. Реализовать всё это, да ещё и эффективно — отнюдь не тривиальная задача. И снова, желая показать всё множество вариаций, мы позволим себе применить для обычных функций слегка необычный, но весьма последовательный во всех смыслах подход (понимая, однако, что его быстродействие оставляет желать лучшего). [**см. упр. 10.1**]

Замыкания будут представляться специальными структурами. Первые два поля содержат указатель на соответствующую функцию и её арность. Остальные хранят значения замкнутых свободных переменных. Макрос `SCM_DefineClosure` определяет соответствующие типы структур Си для всех используемых замыканий.

```
#define SCM_DefineClosure(struct_name, fields) \
  struct struct_name {                         \
      SCM (*behavior)(void);                   \
      long arity;                              \
      fields                                   \
  }
```

Первым аргументом замыканий, вызываемых `SCM_invoke`, являются они сами; так реализуется возможность доступа к замкнутому окружению. Если функция вызывается с неправильным количеством аргументов — это ошибка, о которой необходимо сообщить. Проверка арности перекладывается на `SCM_invoke`, так что функции могут быть уверены в количестве аргументов. Единственным непонятным моментом остаются функции с переменной арностью: здесь, видимо, придётся явно передавать им количество аргументов, так как в Си нет возможности его узнать. Поэтому будем считать, что все функции вторым аргументом принимают количество фактических аргументов вызова, а сами аргументы всегда передаются с помощью специального механизма Си, определённого в заголовочном файле `<stdarg.h>` (ранее

<varargs.h>). Как мы и говорили, это не самое элегантное из возможных решений; подход, используемый для примитивов, гораздо эффективнее.

Аргументы объявляемых функций Си (`self_`, `size_` и `arguments_`) названы так, чтобы их нельзя было спутать с переменными Scheme (вспомните о подчёркиваниях). Единственным интересным моментом остаётся инициализация точечной переменной, собирающая в список все оставшиеся аргументы. Список так список, зачем же ещё существует функция `SCM_list`.

```
#define SCM_DeclareFunction(Cname) \
  SCM Cname(struct Cname *self_,   \
            unsigned long size_, va_list arguments_)

#define SCM_DeclareLocalVariable(Cname, index) \
  SCM Cname = va_arg(arguments_, SCM)

#define SCM_DeclareLocalDottedVariable(Cname, index) \
  SCM Cname = SCM_list(size_ - index, arguments_)

#define SCM_Free(Cname) ((self_)->Cname)
```

И всё же в данном подходе есть что-то хорошее: вы только посмотрите, как просто, понятно и красиво выражаются обращения к свободным переменным!

10.10. Библиотека времени исполнения

Библиотека времени исполнения (runtime library) — это набор функций, написанных на Си. Эти функции реализуют возможности, необходимые для полноценной работы программ, поэтому данная библиотека должна быть подключена ко всякой хоть сколь-нибудь нетривиальной программе. Конечно, здесь мы лишь набросаем её общие черты и не будем реализовывать большую часть утилитарных функций вроде `string-ref` или `close-output-port`. Будут рассмотрены только самые важные, типичные библиотечные функции (в частности, здесь нет функции `SCM_print`, которая используется в генерируемой нами `main`).

10.10.1. Выделение памяти

В Си за управление памятью отвечает лично программист, так что сборщика мусора у нас не будет, потому как его написание с нуля заняло бы слишком много времени. За информацией об этом можете обратиться, например, к [Spi90, Wil92].* Интеграцию с Бёмовским сборщиком мусора [BW88] оставляем вам в качестве упражнения.

* Или к более современной книге: *Richard Jones, Antony Hosking, Eliot Moss.* The Garbage Collection Handbook: The Art of Automatic Memory Management. — Chapman & Hall/CRC, 2012. — 511 p.

Наиболее очевидная функция, требующая выделения памяти, — это `cons`. Она должна создать новую точечную пару, заполнить её поля (включая поле типа) и вернуть адрес созданной пары в виде объекта типа `SCM`.

```
SCM SCM_cons(SCM a, SCM d)
{
    SCMref cell = malloc(sizeof(struct SCM_unwrapped_pair));
    if (!cell)
    {
        SCM_error(SCM_ERR_CANT_ALLOC);
    }

    cell->pair.header.tag = SCM_PAIR_TAG;
    cell->pair.car = a;
    cell->pair.cdr = d;
    return SCM_Wrap(cell);
}
```

Замыкания создаются функцией `SCM_close`. Их арность может быть любой, что обосновывает использование функций переменной арности языка Си. Итак, нам надо создать объект с типом `SCM_CLOSURE_TAG` и заполнить его поля полученными значениями. Порядок перечисления значений должен в точности совпадать с порядком полей.

```
SCM SCM_close(SCM (*Cfunction)(void), long arity,
              unsigned long size, ...)
{
    unsigned long i;
    va_list args;
    SCMref result = malloc(sizeof(struct SCM_unwrapped_closure)
                           + (size - 1) * sizeof(SCM));
    if (!result)
    {
        SCM_error(SCM_ERR_CANT_ALLOC);
    }

    result->closure.header.tag = SCM_CLOSURE_TAG;
    result->closure.behavior   = Cfunction;
    result->closure.arity      = arity;
    va_start(args, size);
    for (i = 0; i < size; i++) {
        result->closure.environment[i] = va_arg(args, SCM);
    }
    va_end(args);

    return SCM_Wrap(result);
}
```

10.10.2. Функции над парами

Функции `car` и `set-car!` очень просты, но не следует их путать с одноимёнными макросами. Данные функции типобезопасны в том смысле, что они проверяют типы аргументов перед тем, как лезть непонятно куда в память. В общем случае у каждой безопасной функции есть небезопасный аналог, который можно использовать только тогда, когда компилятор полностью уверен в типе аргументов. Несмотря на то, что Лисп динамически типизирован, программы на нём обычно таковы, что как минимум две трети проверок типов, по мнению [Hen92a, WC94, Ser94], можно смело убирать. Дело в том, что программисты помнят о типах, выполняют часть проверок у себя в голове и не передают функциям что попало. Хорошие компиляторы осведомлены о существовании «белкового препроцессора» и оптимизируют код с учётом этого обстоятельства.

```
SCM SCM_car(SCM x)
{
    if (SCM_PairP(x)) {
        return SCM_Car(x);
    }
    else return SCM_error(SCM_ERR_CAR);
}

SCM SCM_set_cdr(SCM x, SCM d)
{
    if (SCM_PairP(x)) {
        SCM_Unwrap(x)->pair.cdr = d;
        return x;
    }
    else return SCM_error(SCM_ERR_SET_CDR);
}
```

Неплохо будет также рассмотреть функцию `list`. Она может принимать произвольное число аргументов, поэтому первым её аргументом идёт это самое число. (Понятно, что в Си не стоит записывать итерацию с помощью рекурсии, но соблазн был слишком велик.)

```
SCM SCM_list(unsigned long count, va_list arguments)
{
    if (count == 0) {
        return SCM_nil;
    }
    else {
        SCM car = va_arg(arguments, SCM);
        return SCM_cons(car, SCM_list(count - 1, arguments));
    }
}
```

Возникающие в процессе работы ошибки никак не перехватываются и не обрабатываются. Если возникают какие-либо проблемы, то просто вызывается макрос `SCM_error`. Он сообщает наверх о том, что и где произошло, выводя код ошибки, имя провинившегося файла и номер строки в нём.

```
#define SCM_error(code) SCM_signal_error(code, __LINE__, __FILE__)

SCM SCM_signal_error(unsigned long code,
                     unsigned long line,
                     const char *file)
{
    fflush(stdout);
    fprintf(stderr, "Error %ul, line %ul, file %s.\n", code, line, file);
    exit(code);
}
```

10.10.3. Вызовы функций

Самыми важными (и хитроумными) из функций, вызывающих другие функции, являются `SCM_invoke` и `apply`. Присутствие здесь примитива `apply` не особо удивительно — он же должен знать протокол вызова функций, чтобы ему следовать.

За исключением вызовов, интегрированных непосредственно в код (то есть реализованных через прямые обращения к макросам или функциям на Си), все остальные вызовы выполняются внутренней функцией `SCM_invoke`. Первым аргументом она принимает вызываемую функцию, за которой следуют количество и значения аргументов для неё. Как и в предыдущих интерпретаторах, `SCM_invoke` отвечает за все возможные вызовы. Она анализирует тип первого аргумента, чтобы выбрать правильный метод обработки (или выругаться об ошибке типизации). Затем она извлекает вызываемый объект, выясняет поддерживаемую им арность и сравнивает её с фактической. Наконец, она выполняет собственно вызов, передавая функции аргументы соответствующим образом. В нашем случае есть три различных протокола вызова. Это далеко не исчерпывающий список возможных вариантов; например, вместо указания количества аргументов можно просто дописывать в конец какое-то специальное значение (именно так поступает Bigloo, используя для этой цели `NULL`).

- Примитивы с фиксированной арностью (вроде `SCM_cons`) вызываются просто и без всяких ухищрений. Эти вызовы можно отнести к виду $f(x, y, z)$.
- Примитивы с переменной арностью (вроде `SCM_list`) должны знать количество полученных аргументов. Оно передаётся дополнительным аргументом, поэтому вызов имеет вид $f(n, x_1, x_2, \ldots, x_n)$.

- Замыканиям необходимо получить не только все аргументы вызова, но также и значения свободных переменных, которые фактически хранятся внутри самих замыканий. Поэтому для них надо выполнять вызовы вида $f(f, n, x_1, x_2, \ldots, x_n)$.

Естественно, мы могли бы улучшить эти протоколы, привести их к единому виду, возможно, даже избавиться от некоторых. Но это потом, сейчас у нас есть вполне конкретные требования к `SCM_invoke`. Несмотря на внушительные размеры, она довольно регулярна по своей структуре. (Обработка продолжений будет рассмотрена в следующем разделе.)

```
SCM SCM_invoke(SCM function, unsigned long number, ...)
{
    /* Вряд ли у нас получится вызвать число */
    if (SCM_FixnumP(function)) {
        return SCM_error(SCM_ERR_CANNOT_APPLY);
    }

    switch (SCM_2tag(function)) {
      case SCM_SUBR_TAG: {
        SCM (*behavior)(void) = (SCM_Unwrap(function)->subr).behavior;
        long arity = (SCM_Unwrap(function)->subr).arity;

        /* Фиксированная арность */
        if (arity >= 0) {
            if (arity != number) {
                return SCM_error(SCM_ERR_WRONG_ARITY);
            }
            if (arity == 0) {
                return behavior();
            }
            else {
                SCM result;
                va_list args;
                va_start(args, number);
                switch (number) {
                  case 1: {
                    SCM a0 = va_arg(args, SCM);
                    result = ((SCM (*)(SCM)) behavior)(a0);
                    break;
                  }
                  case 2: {
                    SCM a0 = va_arg(args, SCM);
                    SCM a1 = va_arg(args, SCM);
                    result = ((SCM (*)(SCM, SCM)) behavior)(a0, a1);
                    break;
                  }
```

```
            case 3: {
              SCM a0 = va_arg(args, SCM);
              SCM a1 = va_arg(args, SCM);
              SCM a2 = va_arg(args, SCM);
              result = ((SCM (*)(SCM,SCM,SCM)) behavior)
                       (a0, a1, a2);
              break;
            }
            default:
              /* Не существует примитивов большей арности */
              return SCM_error(SCM_ERR_INTERNAL);
          }
          va_end(args);
          return result;
      }
  }
  /* Переменная арность */
  else {
      if (number < SCM_MinimalArity(arity)) {
          return SCM_error(SCM_ERR_MISSING_ARGS);
      }
      else {
          SCM result;
          va_list args;
          va_start(args, number);
          result = ((SCM (*)(unsigned long, va_list)) behavior)
                   (number, args);
          va_end(args);
          return result;
      }
  }
}

case SCM_CLOSURE_TAG: {
  SCM result;
  va_list args;
  SCM (*behavior)(void) =
          (SCM_Unwrap(function)->closure).behavior;
  long arity = (SCM_Unwrap(function)->closure).arity;

  if (arity >= 0 && number != arity) {
      return SCM_error(SCM_ERR_WRONG_ARITY);
  }
  if (number < SCM_MinimalArity(arity)) {
      return SCM_error(SCM_ERR_MISSING_ARGS);
  }
```

```
          va_start(args, number);
          result = ((SCM (*)(SCM, unsigned long, va_list)) behavior)
                   (function, number, args);
          va_end(args);
          return result;
        }

        /* Невызываемо! */
        default:
          SCM_error(SCM_ERR_CANNOT_APPLY);
      }
  }
```

Реализация примитива `apply` имеет сравнимые размеры. Так как это примитив переменной арности, то свои аргументы он получает в виде `va_list`, содержащего применяемую функцию и список аргументов для неё. Но здесь возникает проблема с тем, что в C90 отсутствует* способ передать другой функции аргументы, оставшиеся в `va_list`. Что ж, придётся с этим смириться и разбирать все возможные арности вручную. Печально, конечно, но нам придётся ограничить максимальное количество аргументов `apply`. Стандартизацией Common Lisp однозначно руководили реалисты, поэтому там определена специальная константа `CALL-ARGUMENTS-LIMIT`, хранящая максимально допустимое количество аргументов. По стандарту она не может быть меньше 50. Мы ограничимся[11] всего 14, из которых здесь покажем обработку лишь первых четырёх.

```
SCM SCM_apply(unsigned long number, va_list arguments)
{
    unsigned long i;
    SCM args[31];
    SCM last_arg;
    SCM fun = va_arg(arguments, SCM);

    for (i = 0; i < number - 1; i++) {
        args[i] = va_arg(arguments, SCM);
    }
    last_arg = args[--i];
    while (SCM_PairP(last_arg)) {
        args[i++] = SCM_Car(last_arg);
        last_arg = SCM_Cdr(last_arg);
    }
    if (!SCM_NullP(last_arg)) {
        SCM_error(SCM_ERR_APPLY_ARG);
    }
```

*В C99 появился макрос `va_copy`, позволяющий сделать это безопасно.

[11] Если вам здесь видится нечеловеческое ущемление прав, то сперва узнайте предел вашей любимой реализации Scheme.

```
    switch (i) {
      case 0: return SCM_invoke(fun, 0);
      case 1: return SCM_invoke(fun, 1, args[0]);
      case 2: return SCM_invoke(fun, 2, args[0], args[1]);
      case 3: return SCM_invoke(fun, 3, args[0], args[1], args[2]);
      /* ... */
      default: return SCM_error(SCM_ERR_APPLY_SIZE);
    }
}
```

Всё равно C90 не поддерживает более 32 параметров функций, так что на 50 в нашем случае можно и не надеяться. Пуристы, наверное, уже давно обратили внимание на (ненужную?) проверку того, что в конце списка аргументов `apply` действительно находится `()`.

Существенной проблемой прямого использования функций Си является то, что Си не гарантирует оптимизацию хвостовых вызовов. Поэтому некоторые программы, без проблем выполняющиеся на Scheme, при буквальном переводе на Си будут умирать от переполнения стека. Некоторые компиляторы Scheme в Си (например, Scheme→C или Bigloo) применяют множество хитроумных алгоритмов, чтобы избежать подобных аварий. Они генерируют специальный код для рекурсии, итерации через рекурсию и так далее, но, к сожалению, абсолютно все варианты предусмотреть невозможно, и обязательно найдётся программа, которая всё сломает. Альтернативным решением является отказ от использования стека Си для представления продолжений Scheme.

10.11. `call/cc`: быть или не быть

Мы только что закончили рассмотрение принципов компиляции основной части конструкций Scheme в программы на Си. Несмотря на то, что многие стандартные функции не реализованы, важнейшие из них всё же присутствуют (хоть, возможно, и не в самом эффективном виде). Все, кроме функций для работы с продолжениями. Для начала мы реализуем функцию `call/ep`, предоставляющую продолжения с динамическим временем жизни, как в Лиспе. Она будет использовать `setjmp`/`longjmp`.

Во второй же части данного раздела мы разберёмся с тем, как реализовать на Си истинно неограниченные продолжения Scheme.

10.11.1. Функция `call/ep`

Функция `call/ep` во всём идентична `call/cc`, за исключением того, что продолжения, создаваемые ею, хоть и являются полноценными объектами Scheme, но не могут быть использованы вне соответствующей формы `call/ep`, которая ограничивает время их жизни временем исполнения собственного

тела. [**см. стр. 130**] Для её реализации применяется подход, аналогичный показанному в седьмой главе, где в стек помещалась специальная метка, служившая точкой прибытия для продолжений. Теперь такой меткой будет локальная переменная типа `jmp_buf`. Вот необходимые для этого функции:

```
SCM SCM_allocate_continuation(struct SCM_jmp_buf *address)
{
    SCMref continuation = malloc(sizeof(struct SCM_unwrapped_escape));
    if (!continuation)
    {
        SCM_error(SCM_ERR_CANT_ALLOC);
    }

    continuation->escape.header.tag    = SCM_ESCAPE_TAG;
    continuation->escape.stack_address = address;
    return SCM_Wrap(continuation);
}

struct SCM_jmp_buf {
    SCM back_pointer;
    SCM jumpvalue;
    jmp_buf xen;
};

SCM SCM_callep(SCM f)
{
    struct SCM_jmp_buf scmjb;
    SCM continuation = SCM_allocate_continuation(&scmjb);
    scmjb.back_pointer = continuation;

    if (setjmp(scmjb.xen)) {
        return scmjb.jumpvalue;
    }
    else {
        return SCM_invoke1(f, continuation);
    }
}
```

Активация продолжений обрабатывается функцией `SCM_invoke`, которая перед выполнением перехода ещё проверит арность вызова. Следующий фрагмент кода, реализующий данную обработку, должен быть добавлен в приведённое ранее определение `SCM_invoke`. Перед выполнением перехода достаточно (ввиду отсутствия форм `unwind-protect`) убедиться в том, что продолжение указывает на соответствующий ему `jmp_buf`, а мы находимся *выше* этого `jmp_buf` на стеке. К сожалению, для второй проверки необходимо знать направление роста стека. Будем считать, что макрос `SCM_STACK_HIGHER` раскрывается в соответствующее отношение порядка. Определение этого макроса, естественно, возлагается на реализацию; благо, нужную информацию легко

получить небольшой программкой на абсолютно портируемом Си. Обычно (по крайней мере в Un⋆x) стек растёт вниз, поэтому можете считать, что SCM_STACK_HIGHER раскрывается в <=.

```
case SCM_ESCAPE_TAG:
  if (number == 1) {
      struct SCM_jmp_buf *address =
          SCM_Unwrap(function)->escape.stack_address;
      if (SCM_EqP(address->back_pointer, function) &&
          ((void*) &address SCM_STACK_HIGHER (void*) address))
      {
          va_list args;
          va_start(args, number);
          address->jumpvalue = va_arg(args, SCM);
          va_end(args);
          longjmp(address->xen, 9);
      }
      else {
          /* Слишком поздно! */
          return SCM_error(SCM_ERR_OUT_OF_EXTENT);
      }
  }
  else {
      return SCM_error(SCM_ERR_MISSING_ARGS);
  }
```

Эффективность call/ep, о которой говорилось при рассмотрении компилятора в байт-код, сейчас несколько приуменьшилась, так как переходы с помощью longjmp печально известны своей «стремительностью». Они неплохи для реализации исключений, но при постоянном использовании существенно замедляют исполнение программ. Однако, мы лишь хотели показать пример интеграции Лиспа и Си, поэтому ничего особо страшного в этом нет.

10.11.2. Функция call/cc

Ура! Если вы скучаете по настоящим продолжениям, то скоро ваши мечты сбудутся. До этого call/cc чаще всего реализовывалась с помощью магии — неявного вызова уже существующей call/cc языка реализации. К сожалению, в Си такой функции нет, так что пришло время *написать* её самостоятельно. Однако для этого необходимо знать, как здесь устроен стек. Проблема в том, что в Си нет переносимого способа взаимодействия с ним. Вернее, нет *простого* способа — всегда можно написать кучу платформоспецифичных вариантов и выбирать из них нужный с помощью #ifdef. Для этой книги такой вариант не подходит, так что попробуем подойти с другой стороны: приведём программу к стилю передачи продолжений (CPS). Тогда и продолжения будут как на ладони, и компилятор останется почти неизменным.

Делаем продолжения явными

Предлагаемое преобразование эквивалентно показанному ранее, только выполняется оно с помощью нового обходчика кода. [**см. стр. 216**] Работа ведётся над уже объектифицированным исходным кодом.

У CPS-преобразования, выполняемого в лоб, есть недостаток: оно очень щедро использует `let`-формы, но при этом не заботится об эффективности и представляет их так называемыми административными редексами. Поэтому после выполнения этого преобразования мы ещё раз пройдёмся по коду, возвращая программу во вменяемый вид. Этим займётся функция `letify`. Итак, ядро компилятора с поддержкой продолжений:

```
(define (compile->C e out)
  (set! g.current '())
  (let* ((ee (letify (cpsify (Sexp->object e)) '()))
         (prg (extract-things! (lift! ee))) )
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )
```

Извлечение продолжений выполняет функция `cpsify`. Она организует совместную работу обходчика `update-walk!` и функции `->CPS`. Также нам понадобятся два новых класса: класс продолжений, чтобы отличать обычные функции от функций-продолжений, и класс псевдопеременных, представляющих временные переменные неявно используемых (в исходном коде на Scheme) продолжений.

```
(define-class Continuation    Function ())
(define-class Pseudo-Variable Local-Variable ())

(define (cpsify e)
  (let ((v (new-Variable)))
    (->CPS e (make-Continuation (list v) (make-Local-Reference v))) ) )

(define new-Variable
  (let ((counter 0))
    (lambda ()
      (set! counter (+ 1 counter))
      (make-Pseudo-Variable counter #f #f) ) ) )
```

Функция `->CPS` первым аргументом принимает объект, представляющий вычисления, а вторым — продолжение, которому надо передать результат этих вычислений. В общем случае ничего кроме этого делать и не надо. Начальным продолжением, которому `cpsify` передаёт результат выполнения всей программы, является тождество $\lambda x.x$.

```
(define-generic (->CPS (e Program) k)
  (convert2Regular-Application k e) )

(define (convert2Regular-Application k . args)
  (make-Regular-Application k (convert2arguments args)) )
```

Теперь остаётся лишь определить методы для всех остальных узлов синтаксического дерева. Последовательные вычисления? Элементарно: вычисляем одну форму сейчас, а остальные передаём продолжению:

```
(define-method (->CPS (e Sequence) k)
  (->CPS (Sequence-first e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (->CPS (Sequence-last e) k) ) ) ) )
```

С ветвлением тоже всё просто, надо только помнить, что обе ветви пользуются одним и тем же исходным продолжением:

```
(define-method (->CPS (e Alternative) k)
  (->CPS (Alternative-condition e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (make-Alternative
                      (make-Local-Reference v)
                      (->CPS (Alternative-consequent e) k)
                      (->CPS (Alternative-alternant e) k) ) ) ) ) )
```

И даже присваивания очевидны. Сначала вычисляется новое значение, продолжением этих вычислений является обновление значения переменной и выполнение всего того, что следует дальше за присваиванием:

```
(define-method (->CPS (e Box-Write) k)
  (->CPS (Box-Write-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Box-Write
                         (Box-Write-reference e)
                         (make-Local-Reference v) ) ) ) ) ) )

(define-method (->CPS (e Global-Assignment) k)
  (->CPS (Global-Assignment-form e)
         (let ((v (new-Variable)))
           (make-Continuation
            (list v) (convert2Regular-Application
                      k (make-Global-Assignment
                         (Global-Assignment-variable e)
                         (make-Local-Reference v) ) ) ) ) ) )
```

А вот введение локальных переменных `let`-формами уже сложнее. Дело в том, что продолжение `k` надо протащить внутрь узлов `Fix-Let` как новую переменную. Самый простой способ этого добиться — превратить все `Fix-Let` обратно в приводимые формы (аппликации непосредственно задаваемых абстракций) на время выполнения CPS-преобразования. Такие вспомогательные формы называются *административными* редексами [SF92]. Хоть они

выглядят не особо красиво и замедляют наивную интерпретацию, это лишь временное промежуточное представление. На эффективность результата компиляции эти (и все остальные) редексы не повлияют — они будут обнаружены и оптимизированы функцией `letify`.

```
(define-method (->CPS (e Fix-Let) k)
  (->CPS (make-Regular-Application
          (make-Function (Fix-Let-variables e) (Fix-Let-body e))
          (Fix-Let-arguments e) )
         k ) )
```

Все функции получают по дополнительному аргументу, представляющему продолжение их вызова.

```
(define-method (->CPS (e Function) k)
  (convert2Regular-Application k
   (let ((k (new-Variable)))
     (make-Function (cons k (Function-variables e))
                    (->CPS (Function-body e)
                           (make-Local-Reference k) ) ) ) ) )
```

Аппликации функций должны выполняться с учётом этого нововведения. Все аргументы последовательно вычисляются перед применением к ним функции. Порядок вычисления обычный: слева направо.

```
(define-method (->CPS (e Predefined-Application) k)
  (let* ((args (Predefined-Application-arguments e))
         (vars (let name ((args args))
                 (if (not (Arguments? args)) '()
                     (cons (new-Variable)
                           (name (Arguments-others args)) ) ) ))
         (application
          (convert2Regular-Application k
           (make-Predefined-Application
            (Predefined-Application-variable e)
            (convert2arguments
             (map make-Local-Reference vars) ) ) ) ) )
    (arguments->CPS args vars application) ) )

(define-method (->CPS (e Regular-Application) k)
  (let* ((fun (Regular-Application-function e))
         (args (Regular-Application-arguments e))
         (varfun (new-Variable))
         (vars (let name ((args args))
                 (if (not (Arguments? args)) '()
                     (cons (new-Variable)
                           (name (Arguments-others args)) ) ) ))
         (application
          (make-Regular-Application
           (make-Local-Reference varfun)
```

```
            (make-Arguments k (convert2arguments
                               (map make-Local-Reference vars)) ) ) )
    (->CPS fun (make-Continuation
                (list varfun)
                (arguments->CPS args vars application)) ) )

(define (arguments->CPS args vars appl)
  (if (pair? args)
      (->CPS (Arguments-first args)
             (make-Continuation
              (list (car vars))
              (arguments->CPS (Arguments-others args)
                              (cdr vars) appl ) ) )
      appl ) )
```

Оптимизируем редексы

Функция `letify`, упомянутая ранее, отвечает за обратное преобразование приводимых форм в `let`-формы. Заодно её можно использовать для исправления других неудобств CPS-преобразования. Когда функция `->CPS` обрабатывает ветвление, она передаёт каждой ветви физически одно и то же продолжение. Из-за этого в итоге получается не дерево, а направленный ациклический граф. Некоторые из последующих преобразований могут не дружить с такими графами, поэтому `letify`, чтобы гарантированно избежать проблем в будущем, скопирует все дуги графа в новое дерево (где разделяемые дуги превратятся в пары одинаковых ветвей). Предположим, у нас для этого есть обобщённая функция `clone`, способная скопировать любой объект Meroonet. [**см. упр. 11.2**] Применение её к переменным отчасти напоминает переименование, выполняемое `collect-temporaries!`.

```
(define-generic (letify (o Program) env)
  (update-walk! letify (clone o) env) )

(define-method (letify (o Function) env)
  (let* ((vars (Function-variables o))
         (body (Function-body o))
         (new-vars (map clone vars)) )
    (make-Function
     new-vars
     (letify body (append (map cons vars new-vars) env)) ) ) )

(define-method (letify (o Local-Reference) env)
  (let* ((v (Local-Reference-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Local-Reference (cdr r))
        (letify-error "Disappeared variable" o) ) ) )
```

```
(define-method (letify (o Regular-Application) env)
  (if (Function? (Regular-Application-function o))
      (letify (process-closed-application
               (Regular-Application-function o)
               (Regular-Application-arguments o) )
              env )
      (make-Regular-Application
       (letify (Regular-Application-function o) env)
       (letify (Regular-Application-arguments o) env) ) ) )

(define-method (letify (o Fix-Let) env)
  (let* ((vars (Fix-Let-variables o))
         (new-vars (map clone vars)) )
    (make-Fix-Let
     new-vars
     (letify (Fix-Let-arguments o) env)
     (letify (Fix-Let-body o)
             (append (map cons vars new-vars) env) ) ) ) )

(define-method (letify (o Box-Creation) env)
  (let* ((v (Box-Creation-variable o))
         (r (assq v env)) )
    (if (pair? r)
        (make-Box-Creation (cdr r))
        (letify-error "Disappeared variable" o) ) ) )

(define-method (clone (o Pseudo-Variable)) (new-Variable))
```

Дорабатываем библиотеку

Возможно, вам не совсем понятно, что мы приобретаем, выполнив данное преобразование. Идея в том, чтобы сделать продолжения явными, — то есть сделать реализацию `call/cc` тривиальной. Продолжения в таком случае являются просто замыканиями, никогда не возвращающими значений. Вот определение `SCM_callcc`. Раз это обычная функция, то первым аргументом она принимает продолжение своего вызова; да, теперь она бинарна. Функция `SCM_invoke_continuation` определяется перед ней, так как компилятору Си не нравится беспорядок.

```
SCM SCM_invoke_continuation(SCM self, unsigned long number,
                            va_list arguments)
{
    SCM current_k = va_arg(arguments, SCM);
    SCM value     = va_arg(arguments, SCM);
    return SCM_invoke1(SCM_Unwrap(self)->closure.environment[0], value);
}
```

```
SCM SCM_callcc(SCM k, SCM f)
{
    SCM reified_k =
        SCM_close(SCM_CfunctionAddress(SCM_invoke_continuation),
                  2, 1, k);
    return SCM_invoke2(f, k, reified_k);
}

SCM_DefinePredefinedFunctionVariable(CALLCC, "CALL/CC", 2, SCM_callcc);
```

Все остальные библиотечные функции остались нетронутыми, а ведь их протокол вызова претерпел некоторые изменения в связи с появлением продолжений. Если CPS-преобразование выдаст что-то вроде `(let ((f car)) (f '(a b)))`, то наш компилятор-дурачок так и не поймёт, что это на самом деле просто вызов примитива `(car '(a b))` (а то и вовсе `(quote a)`, если провести свёртку констант). Он честно вычислит значение глобальной переменной `CAR` и выполнит вызов обычной функции, строго придерживаясь протокола: передав продолжение первым аргументом, а точечную пару — вторым. Значением `CAR` теперь должна быть абстракция `(lambda (k p) (k (car p)))`, определённая через изначальный примитив `car`. С этой целью мы введём несколько вспомогательных макросов и добавим каждой библиотечной функции дополнительный аргумент. Функции с приставкой `SCMq_` являются новым интерфейсом функций, начинающихся на `SCM_`.

```
#define SCM_DefineCPSsubr2(new_name, old_name) \
  SCM new_name(SCM k, SCM x, SCM y)            \
  {                                            \
      return SCM_invoke1(k, old_name(x, y));   \
  }

#define SCM_DefineCPSsubrN(new_name, old_name)                \
  SCM new_name(unsigned long number, va_list arguments)       \
  {                                                           \
      SCM k = va_arg(arguments, SCM);                         \
      return SCM_invoke1(k, old_name(number - 1, arguments)); \
  }

SCM_DefineCPSsubr2(SCMq_gtp, SCM_gtp)
SCM_DefinePredefinedFunctionVariable(GREATERP, " > ", 3, SCMq_gtp);

SCM_DefineCPSsubrN(SCMq_list, SCM_list)
SCM_DefinePredefinedFunctionVariable(LIST, "LIST", -2, SCMq_list);
```

Единственной проблемной функцией остаётся `apply`. Она использует протокол вызова напрямую, так что о появившихся продолжениях ей придётся рассказывать лично:

```
SCM SCMq_apply(unsigned long number, va_list arguments)
{
    unsigned long i;
    SCM args[30];
    SCM last_arg;
    SCM k   = va_arg(arguments, SCM);
    SCM fun = va_arg(arguments, SCM);

    for (i = 0; i < number - 2; i++) {
        args[i] = va_arg(arguments, SCM);
    }
    last_arg = args[--i];
    while (SCM_PairP(last_arg)) {
        args[i++] = SCM_Car(last_arg);
        last_arg = SCM_Cdr(last_arg);
    }
    if (!SCM_NullP(last_arg)) {
        SCM_error(SCM_ERR_APPLY_ARG);
    }

    switch (i) {
      case 0: return SCM_invoke(fun, 1, k);
      case 1: return SCM_invoke(fun, 2, k, args[0]);
      case 2: return SCM_invoke(fun, 3, k, args[0], args[1]);
      case 3: return SCM_invoke(fun, 3, k, args[0], args[1], args[2]);
      /* ... */
      default: return SCM_error(SCM_ERR_APPLY_SIZE);
    }
}
```

Пример

Давайте возьмём наш подопытный пример и посмотрим, что же выдаст новый компилятор. Хоть мы и получаем в итоге код на Си, в нём безошибочно узнаётся стиль передачи продолжений.

o/chap10kex.c

```
/* Compiler to C $Revision: 4.1$
(BEGIN
  (SET! INDEX 1)
  ((LAMBDA
     (CNTER . TMP)
     (SET! TMP (CNTER (LAMBDA (I) (LAMBDA X (CONS I X)))))
     (IF CNTER (CNTER TMP) INDEX))
    (LAMBDA (F) (SET! INDEX (+ 1 INDEX)) (F INDEX))
    'FOO))
*/
```

```
#include "scheme.h"

/* Global environment: */
SCM_DefineGlobalVariable(INDEX, "INDEX");

/* Quotations: */
#define thing3 SCM_nil                          /* () */
SCM_DefineString(thing4_object, "FOO");
#define thing4 SCM_Wrap(&thing4_object)
SCM_DefineSymbol(thing2_object, thing4);    /* FOO */
#define thing2 SCM_Wrap(&thing2_object)
#define thing1 SCM_Int2Fixnum(1)
#define thing0 thing1                           /* 1 */

/* Functions: */
SCM_DefineClosure(function_0, SCM I; );

SCM_DeclareFunction(function_0)
{
    SCM_DeclareLocalVariable(v_25, 0);
    SCM_DeclareLocalDottedVariable(X, 1);
    SCM v_27; SCM v_26;
    return (v_26 = SCM_Free(I),
            (v_27 = X,
             SCM_invoke1(v_25,
                         SCM_cons(v_26, v_27))));
}

SCM_DefineClosure(function_1, );

SCM_DeclareFunction(function_1)
{
    SCM_DeclareLocalVariable(v_24, 0);
    SCM_DeclareLocalVariable(I, 1);
    return SCM_invoke1(v_24,
                       SCM_close(SCM_CfunctionAddress(function_0),
                                 -2, 1, I));
}

SCM_DefineClosure(function_2, SCM v_15; SCM CNTER; SCM TMP; );

SCM_DeclareFunction(function_2)
{
    SCM_DeclareLocalVariable(v_21, 0);
    SCM v_20; SCM v_19; SCM v_18; SCM v_17;
```

```
    return (v_17 = (SCM_Content(SCM_Free(TMP)) = v_21),
            (v_18 = SCM_Free(CNTER),
             ((v_18 != SCM_false)
              ? (v_19 = SCM_Free(CNTER),
                 (v_20 = SCM_Content(SCM_Free(TMP)),
                  SCM_invoke2(v_19,
                              SCM_Free(v_15),
                              v_20)))
              : SCM_invoke1(SCM_Free(v_15),
                            SCM_CheckedGlobal(INDEX)))));
}

SCM_DefineClosure(function_3, );

SCM_DeclareFunction(function_3)
{
    SCM_DeclareLocalVariable(v_15, 0);
    SCM_DeclareLocalVariable(CNTER, 1);
    SCM_DeclareLocalVariable(TMP, 2);
    SCM v_23; SCM v_22; SCM v_16;
    return (v_16 = (TMP = SCM_allocate_box(TMP)),
            (v_22 = CNTER,
             (v_23 = SCM_close(SCM_CfunctionAddress(function_1), 2, 0),
              SCM_invoke2(v_22,
                          SCM_close(SCM_CfunctionAddress(function_2),
                                    1, 3, v_15, CNTER, TMP),
                          v_23))));
}

SCM_DefineClosure(function_4, );

SCM_DeclareFunction(function_4)
{
    SCM_DeclareLocalVariable(v_8, 0);
    SCM_DeclareLocalVariable(F, 1);
    SCM v_11; SCM v_10; SCM v_9; SCM v_12; SCM v_14; SCM v_13;
    return (v_13 = thing1,
            (v_14 = SCM_CheckedGlobal(INDEX),
             (v_12 = SCM_Plus(v_13, v_14),
              (v_9 = (INDEX = v_12),
               (v_10 = F,
                (v_11 = SCM_CheckedGlobal(INDEX),
                 SCM_invoke2(v_10,
                             v_8,
                             v_11)))))));
}
```

```
SCM_DefineClosure(function_5, );

SCM_DeclareFunction(function_5)
{
    SCM_DeclareLocalVariable(v_1, 0);
    return v_1;
}

SCM_DefineClosure(function_6, );

SCM_DeclareFunction(function_6)
{
    SCM v_5; SCM v_7; SCM v_6; SCM v_4; SCM v_3; SCM v_2; SCM v_28;
    return (v_28 = thing0,
            (v_2 = (INDEX = v_28),
             (v_3 = SCM_close(SCM_CfunctionAddress(function_3), 3, 0),
              (v_4 = SCM_close(SCM_CfunctionAddress(function_4), 2, 0),
               (v_6 = thing2,
                (v_7 = thing3,
                 (v_5 = SCM_cons(v_6, v_7),
                  SCM_invoke3(v_3,
                              SCM_close(SCM_CfunctionAddress(function_5),
                                        1, 0),
                              v_4,
                              v_5))))))));
}

/* Expression: */
int main(void)
{
    SCM_print(SCM_invoke0(SCM_close(
        SCM_CfunctionAddress(function_6), 0, 0)));

    return 0;
}

/* End of generated code */
```

Как видите, размер получаемого файла вырос с 80 строк до 140. Функций стало в два раза больше (шесть вместо трёх), да и локальных переменных им теперь требуется не 2, а 22. Короче говоря, программа слегка растолстела. И работать стала примерно в полтора раза медленнее. После того, как из `main` был убран вызов `SCM_print`, вычисления обёрнуты в цикл на 10 000 итераций, а программа скомпилирована с ключом `-O1`, время работы CPS-версии по сравнению с оригинальной увеличилось с 1,1 секунды до 1,7. Памяти обоим

вариантам требуется одинаковое количество, но в первом случае эта память берётся из стека (где аппаратная поддержка и компилятор Си творят чудеса), тогда как явные продолжения создаются в куче, что не только само по себе медленно, но и плохо сказывается на работе кеша. Хотя, как показано в [AS94], от большей части недостатков такого подхода вполне можно избавиться.

```
% gcc -ansi -pedantic chap10kex.c scheme.o schemeklib.o
% time a.out
(2 3)
0.000u 0.020s 0:00.00 0.0% 3+3k 0+0io 0pf+0w
% size a.out
text    data    bss     dec     hex
32768   4096    32      36896   9020
```

Рассмотренное преобразование само по себе никак не оптимизирует хвостовую рекурсию, а значит, возможность переполнения стека никуда не делась. Фактически, CPS даже усугубляет ситуацию тем, что здесь постоянно вызываются функции, но ни одна из них никогда не возвращает значения. Функции вызываются, стек растёт, но все `return` выполняются последними, так что уменьшаться он будет лишь в самом работы программы. Вряд ли не особо большой стек Си продержится до этого момента. Одно из возможных решений показано в [Bak]: просто делать время от времени `longjmp`, чтобы избавиться от уже ненужных записей активаций.

10.12. Взаимодействие с Си

Так как наш небольшой компилятор использует структуры и функции языка Си для представления данных и программ, то генерируемый им код способен без особых проблем взаимодействовать с другим кодом на Си. Наличие *внешнего интерфейса* (foreign function interface) чрезвычайно важно для любого языка программирования, так как это позволяет получить доступ к уже написанным и отлаженным библиотекам на других языках. Особенно это касается языка Си, являющегося стандартом де-факто на интерфейсы библиотек. Рассмотрим пример использования подобного интерфейса, чтобы вы осознали его полезность, а также трудности, возникающие в процессе реализации.

Стандартная функция `system` принимает строку, передаёт её командному интерпретатору операционной системы и возвращает обратно код возврата, полученный в результате выполнения команд из переданной строки. Предположим, у нас есть макрос `defforeignprimitive`, позволяющий определить для компилятора интерфейс этой функции:

```
(defforeignprimitive system int ("system" string) 1)
```

Теперь компилятор будет понимать форму `(system` π`)` следующим образом: сначала выполняется проверка, что π это действительно строка, затем

вызывается функция `system`, после чего возвращённое ею значение (типа `int`) преобразуется в целое число Scheme и становится значением всей формы. Так как строки Scheme и Си представляются почти одинаково, то это преобразование выполняется легко, чего не скажешь, правда, о других возможных типах данных. Проблемы *маршалинга* (так называются данные преобразования) подробнее рассматриваются в [RM92, DPS94a].

Конечно, одних преобразований типов недостаточно для полной интеграции языков. Функция `system` должна стать полноценным объектом Scheme. Должно быть возможным вызвать её с помощью `apply` или, например, сохранить в переменной. Всё это подразумевает существование некоторого класса значений, представляющих внешние функции.

10.13. Заключение

В данной главе был рассмотрен компилятор из Scheme в Си. Мы написали собственную библиотеку времени исполнения, но с равным успехом можно было использовать любую другую, например, библиотеку Bigloo [Ser94], SIOD [Car94] или SCM [Jaf94]. В процессе мы обсудили проблемы, возникающие при компиляции в язык высокого уровня, различия между Scheme и Си, а также преимущества разумного сотрудничества двух языков.

Читателю настоятельно рекомендуется сравнить полученный компилятор с компилятором в байт-код. Кроме того, остаётся ещё множество других идей, стоящих реализации, например:

- использование `read` для чтения и создания цитат;
- отказ от родного стека Си (и его протокола вызова функций) в пользу независимого стека, заточенного под Scheme;
- поддержка раздельной компиляции, модулей и т. д. [**см. стр. 269**]

Также можно было бы оценить стоимость (в плане увеличения размера библиотеки и уменьшения скорости работы) реализации динамических аспектов языка: функции `load` или её упрощённого аналога — `eval`. Пусть это останется (трудным) упражнением для читателя. Для тех же, кто с ним справится, припасено ещё одно: скомпилировать полученный компилятор самим собой.

10.14. Упражнения

Упражнение 10.1 Вызовы замыканий можно ускорить, адаптировав для них протокол вызова предопределённых примитивов с фиксированной арностью: то есть вызывать их как $f(f, x, y)$. Доработайте компилятор соответствующим образом.

Упражнение 10.2 Доступ к глобальным переменным был бы эффективнее, если бы при чтении не было обязательных проверок на инициализированность, выполняемых `SCM_CheckedGlobal`. Придумайте, как выделить все переменные, для которых эти проверки не требуются, потому что можно быть уверенным в том, что они уже имеют некоторое значение.

Проект 10.3 Рассмотренный в этой главе компилятор преобразует дерево объектов в код на процедурном языке Си. Адаптируйте его для генерации кода на объектно-ориентированном Си++.

Проект 10.4 А теперь попробуйте написать компилятор в специализированный низкоуровневый язык для кодогенерации — Си−−.

Рекомендуемая литература

В последнее время наблюдается существенный интерес к компиляции языков высокого уровня в низкоуровневый Си. Ницан Сеньяк отлично раскрыл эту тему в своей диссертации [Sén91], равно как и Мануэль Серрано в статье [Ser93]. Если вы уже успели влюбиться по уши в подобный подход, то будьте осторожны с исходным кодом Bigloo [Ser94] — можно ведь и не вынырнуть!

Глава 11

Квинтэссенция объектной системы

Объекты! Ах, что бы мы без них делали? В этой главе рассматривается реализация объектной системы, повсеместно используемой в данной книге. Мы намеренно разберём лишь часть её возможностей, дабы не перегружать изложение несущественными деталями. Действительно, хотелось бы ограничиться тем, что Франсуа Рабле называл *substantifique moelle*, — сердцевиной объектной системы, её квинтэссенцией.

Эта объектная система называется Meroon.[1] Подобные системы весьма сложны и требуют существенных усилий, чтобы оставаться одновременно эффективными и переносимыми. Как результат, её архитектура во многом продиктована, а кое-где и искажена тревогами о переносимости. Поэтому здесь мы будем рассматривать более аккуратную уменьшенную версию Meroon, называемую Meroonet.[2]

Лисп уже долгое время идёт рука об руку с объектами. Именно на Лиспе изначально был реализован Smalltalk — один из первых объектно-ориентированных языков. С тех пор Лисп, являясь прекрасным инструментом разработки, послужил колыбелью для множества исследований на тему объектов. Упомянем лишь пару из них: Flavors, которая разрабатывалась в компании Symbolics как GUI-фреймворк, давшая миру идею множественного наследования; Loops, созданная в Xerox PARC, благодаря которой появились обобщённые функции. Кульминацией работ в этих направлениях являются CLOS (Common Lisp Object System) и ΤΕΛΟΣ (The EuLisp Object System). Они объединяют большинство возможностей своих предшественников, но главной их заслугой является введение объектов в систему типов языка. Таким образом эти системы стали истинно объектно-ориентированными — в них всё состоит из объектов.

По сравнению с другими языками, для объектных систем Лиспа характерны следующие две особенности:

[1] Вообще так звали плюшевого мишку моего сына, но если хотите, можете считать это имя акронимом и придумать ему какую-нибудь расшифровку.

[2] Полные исходные коды обеих систем — Meroon и Meroonet — можно найти на сервере, адрес которого приведён на странице 17.

- Обобщённые функции, а также мультиметоды, реализующие механизм *множественной диспетчеризации*.

  Отправка сообщения записывается как (*сообщение аргументы*...), что синтаксически идентично вызову функции: (*функция аргументы*...). Мультиметоды же не предполагают, что получателем сообщения будет один-единственный объект, хоть чаще всего это именно так. Вместо этого методы считаются функциями классов существенных аргументов (дискриминантов). Например, вывод в поток числа в шестнадцатеричном виде не является методом исключительно чисел или только потоков, поведение данного метода определяется декартовым произведением типов: число × поток.

- Рефлексия.

  Рефлексия — это способность системы осознавать собственную структуру. Такие системы реализуют понятие *метаклассов*. Классы объектов в них сами являются полноценными объектами — экземплярами классов, называемых метаклассами, которые, в свою очередь, тоже являются объектами, и так далее. Спецификация этих метаобъектов именуется *метаобъектным протоколом*. Им определяется широта рефлексивных возможностей данной объектной системы: например, от простой интроспекции до полноценной самомодификации. Всё это даёт возможность точного и обратимого представления любых объектов в виде строк байтов, что необходимо для сохранения их в файлах, базах данных или передачи по сети. Кроме того, интроспекция чрезвычайно полезна при разработке как инструмент отладки кода, может существенно помочь компиляторам [**см. стр. 405**], а также незаменима в распределённых системах [Que94].

Нашим же вкладом в историю ООП в Лиспе станет MEROONET. Будем надеяться, что поставленные ограничения не оставят в тени отличные возможности данной системы. Среди них:

- Объектами MEROONET можно выразить любые значения Scheme (включая векторы) без каких-либо ограничений на наследование.

- MEROONET — это рефлексивная объектная система, где каждый класс является отдельным полноценным объектом, полностью доступным для изучения. Проблема бесконечной регрессии решается подобно ObjVlisp [Coi87, BC87].

- Поддерживаются обобщённые функции *а-ля* CLOS [BDG+88], но без мультиметодов.

- Очень эффективная реализация перечисленных возможностей.

Существует множество разнообразных объектных систем для семейства Лиспа в общем и Scheme в частности, например, описанные в [AR88, Kes88, Coi87, MNC+89, Del89, KdRB92]. MEROONET отличается от них в нескольких аспектах:

- Как было сказано ранее, MEROONET реализует обобщённые функции Common Loops [BKK+86], но не мультиметоды.
- Как и ObjVlisp, MEROONET представляет классы полноценными объектами, что положительно сказывается на возможностях рефлексии.
- MEROONET презирает множественное наследование! Как только его семантика будет лучше изучена, а сформулированные в [DHHM92] проблемы — решены, тогда MEROONET пересмотрит свою позицию. Возможно.
- Объекты MEROONET непрерывны, то есть представляются векторами. Такая форма позволяет единообразно выражать любые типы данных, не привязываясь к конкретному языку и его конструкциям [QC88, Que95, Que90a].

Помимо собственно описания MEROONET, мы также обсудим причины, по которым было принято то или иное архитектурное решение. Некоторые из них продиктованы выбором Scheme в качестве языка реализации. Эти решения, конечно же, могли бы быть другими, если бы MEROONET разрабатывалась, к примеру, для Си. Желая упростить повествование, мы будем рассматривать реализацию снизу вверх, вводя новые функции по мере необходимости. Описание возможностей MEROONET здесь приводится вместе с деталями их реализации, но вы также можете пользоваться краткой документацией из третьей главы. [**см. стр. 114**]

11.1. Основы

Первое архитектурное решение касается способа представления объектов MEROONET. Для этого были выбраны непрерывные наборы значений. Такие структуры данных в Scheme называются векторами. Первый элемент вектора (с индексом ноль) будет хранить идентификатор класса, что позволяет легко определить, какому классу принадлежит любой рассматриваемый объект. Наиболее очевидная реализация такой связи — это прямая ссылка на соответствующий класс (на самый конкретный из классов, которым принадлежит объект), но вместо этого мы пронумеруем классы и будем ссылаться на них по этим номерам. Такой подход упростит вывод объектов на экран с помощью стандартных средств Scheme, так как классы — это весьма громоздкие объекты, чья внутренняя структура не особо интересна в данном

случае.[3] Также классы могут содержать циклические ссылки, а стандартная функция `display` с ними не во всех реализациях дружит и может выводить их, мягко говоря, странно. Позже, при рассмотрении обобщённых функций и предикатов принадлежности, появятся и иные причины для предпочтения чисел в качестве идентификаторов классов.

Однако, если классы пронумерованы, то нам однозначно понадобится способ получения нужного класса по его номеру. Для этого все классы помещаются в вектор. К сожалению, в Scheme (в отличие от Common Lisp) векторы не расширяются автоматически, поэтому для простоты мы формально ограничим максимально возможное количество классов, однако явно контролировать фактическое соблюдение установленного ограничения не будем. Переменная `*class-number*` будет хранить индекс первого элемента в векторе `*classes*`, который ещё не занят каким-нибудь классом. Пока что это ноль, но Meroonet использует некоторое количество классов для своих нужд, поэтому со временем начальное значение `*class-number*` увеличится. Кроме того, внутренний код Meroonet можно считать абсолютно корректным, поэтому `number->class` не будет проверять, действительно ли переданное ей значение является допустимым идентификатором класса.

```
(define *class-number* 0)
(define *maximal-number-of-classes* 100)
(define *classes* (make-vector *maximal-number-of-classes* #f))

(define (number->class n)
  (vector-ref *classes* n) )
```

Обращаться к классам по номерам как-то невежливо, гораздо удобнее будет дать им всем имена. Таким образом, анонимных классов не существует. Более того, их имена должны быть известны статически, то есть классы нельзя динамически создавать на лету. Функция `->Class` преобразует символ в одноимённый класс. И снова, для простоты в ней используется банальный линейный поиск, что приводит к интересному побочному эффекту: если определить два класса с одинаковыми именами, то `->Class` гарантированно вернёт более новый из них. Это позволяет в некотором смысле переопределять классы, но это «слабое» переопределение, поэтому без веской причины лучше так не делать.

```
(define (->Class name)
  (let scan ((index (- *class-number* 1)))
    (and (>= index 0)
         (let ((c (vector-ref *classes* index)))
           (if (eq? name (Class-name c))
               c
               (scan (- index 1)) ) ) ) ) )
```

[3] Кроме того такой подход немного помогает сборщику мусора: ему теперь надо проверять и запоминать на одну ссылку меньше.

Обобщённые функции уже *обязаны* иметь имена. Они будут храниться в списке `*generics*`. Функция `->Generic` преобразует символ в соответствующую обобщённую функцию. (Причины такого решения обсудим чуть позже.)

```
(define *generics* (list))

(define (->Generic name)
  (let lookup ((l *generics*))
    (if (pair? l)
        (if (eq? name (Generic-name (car l)))
            (car l)
            (lookup (cdr l)) )
        #f ) ) )
```

11.2. Представление объектов

Как было сказано ранее, объекты MEROONET представляются векторами. Мы хотим иметь возможность представить любое значение Scheme в виде объекта MEROONET, а для этого придётся что-то делать с самими векторами.[4] Предлагается следующее решение: ввести индексированные поля, хранящие несколько значений вместо обычного одного. Эта идея была реализована ещё в Smalltalk [GR83], но там возникают некоторые трудности с наследованием от классов, содержащих индексированные поля. MEROONET не имеет подобного недостатка.

Ещё одна проблема касается строк: ради эффективности они обычно считаются примитивным типом данных. Тем не менее, принципиально их всё же можно понимать как векторы символов — и не то, чтобы это было такой уж плохой идеей в свете Юникода, где один символ может занимать вплоть до четырёх байтов. Однако, в Scheme строкам нельзя добавить новые поля, так что неэффективные явные векторы остаются единственным возможным вариантом представления строк в виде объектов MEROONET.

В MEROONET поля бывают двух видов: обычные (представляемые объектами класса `Mono-Field`) и индексированные (им соответствует класс `Poly-Field`). Обычное поле является просто элементом вектора-объекта, тогда как индексированное реализуется подобно строкам в Паскале: сначала идёт количество элементов этого поля, затем собственно элементы. Рассмотрим наглядный пример. Предположим, определён класс точек:

```
(define-class Point Object (x y))
```

Далее, пусть классу `Point` соответствует номер 7; тогда значение точки, созданной формой `(make-Point 11 22)`, представляется вектором `#(7 11 22)`.

[4] Конечно, как вариант, можно было бы схитрить и воспользоваться тем, что любой объект MEROONET — это уже фактически вектор, только из-за ссылки на класс в начале следует вносить небольшую поправку при индексации.

Многоугольник можно определить как последовательность точек, представляющих его вершины, перечисленные в некотором порядке обхода. Если сделать класс `Polygon` наследником `Point`, то дополнительную пару координат можно понимать как некую опорную точку фигуры (центр, например).

```
(define-class Polygon Point ((* side)))
```

Это пример расширенного синтаксиса описания полей, в таком случае их имена записываются в скобках. Перед обычными полями ставится знак равенства, а перед индексированными — звёздочка.[5] Если нам понадобятся цветные многоугольники, то соответствующий класс можно определить вот так:

```
(define-class ColoredPolygon Polygon ((= color)))
```

Каждое определение класса формирует для него личную армию функций, главной среди которых является функция-конструктор, создающая соответствующие объекты. Её имя состоит из префикса `make-` и имени класса. Каждый создаваемый объект должен иметь чётко определённое начальное состояние, то есть значения всех полей. Для индексированных полей необходимо также указать их размер, который записывается перед перечислением соответствующих значений. Рассмотрим, к примеру, создание оранжевого треугольника (многоугольника с тремя вершинами) и получаемый в результате вектор:

```
(make-ColoredPolygon
  11                      ; x
  22                      ; y
  3 (make-Point 44 55)    ; 3 стороны
    (make-Point 66 77)
    (make-Point 88 99)
  'orange )               ; цвет

→ #(9             ; (Class-number ColoredPolygon-class)
      11          ; x
      22          ; y
      3           ; длина side
      #(7 44 55)  ; side[0]
      #(7 66 77)  ; side[1]
      #(7 88 99)  ; side[2]
      orange )    ; color
```

Любой объект MEROONET — это вектор, в котором первый элемент содержит идентификатор класса данного объекта. Настало время немного упорядочить терминологию; номер элемента вектора-объекта будем называть *смещением*, а к элементам индексированных полей будем обращаться по *индексам*. Итак, в объектах как минимум один элемент зарезервирован под внутренние

[5] Звёздочка используется в качестве символа повторения, как это принято в нотации регулярных выражений, где данный символ означает замыкание Клини.

нужды системы, поэтому смещения полей класса будут начинаться со значения `*starting-offset*`. Тип объекта всегда можно узнать с помощью функции `object->class`.[6] Выяснить, является ли некоторое значение объектом, помогает предикат `Object?`. К сожалению, Scheme не позволяет определять новые типы данных, отличные от существующих. Можно хитрить и мудрить, но все составные значения в конечном итоге принципиально остаются векторами либо списками. Поэтому `Object?` успешно распознаёт все объекты MEROONET, но при этом считает таковыми также любые векторы, начинающиеся на целое число.

```
(define *starting-offset* 1)

(define (object->class o)
  (vector-ref *classes* (vector-ref o 0)) )

(define (Object? o)
  (and (vector? o)
       (integer? (vector-ref o 0)) ) )
```

Завершим данное вступление, пожалуй, перечнем обозначений, которые будут использоваться в этой главе:

`o`	объекты
`v`	значения
`i`	индексы

11.3. Определение классов

Классы определяются с помощью формы `define-class`, принимающей три аргумента:

1) имя определяемого класса;

2) имя его суперкласса;

3) перечень собственных полей.

Создаваемый класс наследует все поля своего суперкласса, а также любые методы обобщённых функций, определённые для него. В некоторых языках состояние (поля) и поведение (методы) класса могут наследоваться отдельно друг от друга.

Синтаксис формы `define-class` следующий:

`(define-class` *имя-класса имя-суперкласса* `(`*поля*`...))`

[6] Эта функция аналогична функции `class-of` из COMMON LISP, EULISP и ISLISP. Как обычно, мы назвали её по-другому, чтобы избежать ложных ассоциаций и позволить использовать несколько объектных систем одновременно.

Поле в списке полей обозначается или просто своим именем (в таком случае это будет обычное поле), или списком из имени и знака равенства (для обычных полей) или звёздочки (для индексированных).

Кроме того, при определении класса автоматически создаются несколько сопутствующих функций. (Надеюсь, они вам понравятся!)

- *Предикат*, распознающий объекты данного класса. Его имя, как принято в Scheme, состоит из имени класса и знака вопроса.
- *Аллокатор*, создающий новые экземпляры класса, но не инициализирующий поля. Их начальные значения могут быть абсолютно любыми. Размеры индексированных полей при этом всё равно должны быть известны, поэтому аллокатор принимает столько аргументов-чисел, сколько у класса индексированных полей. Имя аллокатора состоит из префикса `allocate-` и имени класса.
- *Конструктор*, который создаёт объекты класса с инициализированными полями. Значениям, составляющим индексированное поле, предшествует их количество. Имя конструктора состоит из префикса `make-` и имени класса.
- *Аксессоры* для доступа к обычным и индексированным полям объектов. Каждому полю выдаётся один аксессор чтения (также называемый селектором или геттером), чьё имя состоит из имени класса и имени поля, разделённых дефисом. Соответствующий аксессор записи (сеттер или мутатор) называется аналогично, только начинается на `set-` и, как обычно, заканчивается на восклицательный знак, чтобы подчеркнуть тот факт, что он изменяет состояние памяти. Аксессоры индексированных полей принимают дополнительный аргумент — индекс.
- Вдобавок ко всему, для каждого индексированного поля класса создаётся функция, чьё имя составлено из имени селектора и суффикса `-length`. Очевидно, она возвращает длину соответствующего поля.

Для поддержки рефлексии определяемый класс сам является объектом класса `Class`, представленным в программе глобальной переменной с именем, состоящим из имени класса и суффикса `-class`. В качестве примера рассмотрим функции и переменные,[7] генерируемые формой `(define-class ColoredPolygon Point (color))`.

```
(ColoredPolygon? o)                                  → логическое значение
(allocate-ColoredPolygon sides-number)                  → многоугольник
(make-ColoredPolygon x y sides-number sides... color) → многоугольник
```

[7] Несмотря на то, что индексированные поля состоят из нескольких значений, имена им лучше давать в единственном числе. Тогда выражения вида `(ColoredPolygon-side` *o i*`)` читаются естественнее: «*i*-я сторона цветного многоугольника *o*». Слово `sides` выглядело бы тут странно, ведь мы обращаемся к одной стороне, а не нескольким.

```
(ColoredPolygon-x o)                                  → значение
(ColoredPolygon-y o)                                  → значение
(ColoredPolygon-side o index)                         → значение
(ColoredPolygon-color o)                              → значение

(set-ColoredPolygon-x! o value)                       → не определено
(set-ColoredPolygon-y! o value)                       → не определено
(set-ColoredPolygon-side! o value index)              → не определено
(set-ColoredPolygon-color! o value)                   → не определено

(ColoredPolygon-side-length o)                        → число

ColoredPolygon-class                                  → класс
```

Специальная форма `define-class`, создающая классы, реализуется в виде макроса; к сожалению, со всеми вытекающими из этого последствиями. Первым из них является то, что макрос `define-class` обладает внутренним состоянием — иерархией наследования, — но макросистема Scheme стандарта R^5RS такого не позволяет. Поэтому предполагается, что существует макрос `define-meroonet-macro`, который может без ограничений определять всё необходимое. Этот *интерфейс* является единственной непереносимой частью MEROONET, которую приходится писать вручную для каждой реализации Scheme.

А так ли необходимо глобальное состояние для `define-class`? Ответ: да; для MEROONET в текущем виде, по крайней мере. И вот почему: вместе с классом создаётся набор функций-аксессоров. При создании `Point` автоматически определяются функции `Point-x` и `Point-y` для доступа к полям `x` и `y`. Когда же мы определяем его наследника `Polygon`, MEROONET создаёт для этих полей новые аксессоры `Polygon-x` и `Polygon-y` вместо того, чтобы просто оставить пользователю старые родительские. Определение `Polygon` не содержит описания полей своего суперкласса, об их количестве и именах можно узнать только из определения самого класса `Point`. Следовательно, для правильной работы макроса `define-class` требуется вести учёт и где-то хранить родственные связи классов — это и есть то самое внутреннее состояние.

Данного неудобства можно было бы избежать, применив другой подход к именованию аксессоров. Например, если не упоминать в них имя класса: называть аксессоры поля `x` просто `get-x` и `set-x!`. В этом случае определение класса `Polygon` будет вынуждено модифицировать определение функции `get-x`, чтобы она знала, где в экземплярах `Polygon` лежит поле `x`. Наиболее простой способ реализации такого поведения — это сделать аксессоры обобщёнными функциями, для которых определяются методы всех классов, имеющих соответствующие поля. Так сделано, например, в CLOS, где при определении поля (слота) можно указать имя обобщённой функции, которой будет добавлен метод-геттер (или сеттер, или оба сразу). В таком случае, очевидно, обобщённые функции будут изменяемыми, что не совсем удобно при компиляции из-за усложнения статического анализа кода и проведения оптимизаций.

Поэтому было решено сделать аксессоры обычными неизменяемыми функциями, что гарантирует возможность оптимизаций вроде инлайнинга. Конечно, такое решение тоже не лишено недостатков. Во-первых, функции `Point-x` и `Polygon-x` не обязательно эквивалентны. По сути, они обе извлекают одно и то же поле `x`, но ведь первая делает это для точек, а вторая — для многоугольников. Кажется логичным требовать, чтобы форма `(Polygon-x (make-Point 11 22))` вызывала ошибку типизации, поэтому аксессоры `Point-x` и `Polygon-x` должны быть разными. Кроме философских затруднений, у данного решения есть и более прагматичное следствие: такой подход генерирует много глобальных переменных и функций. Это может огорчить компьютеры с небольшим объёмом памяти, но программы пишутся в первую очередь для людей, а у нас нет особых проблем с запоминанием любого количества фактов, если они тщательно систематизированы, подобно именам наших сопутствующих функций.

Знание полей суперклассов (инкапсулированное во внутреннем состоянии макроса `define-class`) полезно не только при создании аксессоров: если компилятору статически известны количество и порядок всех полей, то он может генерировать более эффективный код для аллокаторов и конструкторов. Чуть позже мы обсудим это в деталях.

В свете рассмотренных соображений, для MEROONET применяется следующее решение: определение класса приводит к созданию экземпляра класса `Class` и размещению данного объекта в глобальной иерархии классов. Всё это будет делать функция `register-class`. Сопутствующие функции будут генерироваться функцией `Class-generate-related-names`. Итого, `define-class` можно определить примерно следующим образом:

```
(define-meroonet-macro (define-class name super-name
                                     own-fields )
  (let ((class (register-class name super-name own-fields)))
    (Class-generate-related-names class) ) )
```

Такое определение ведёт себя не совсем верно при компиляции, так как здесь смешиваются этапы макрораскрытия и исполнения программ. Класс создаётся и вводится в иерархию наследования во время раскрытия макросов, тогда как сопутствующие функции создаются во время исполнения раскрытого кода. Предположим, мы компилируем файл, содержащий определение класса `Polygon`. В конечном итоге получается файл с расширением `*.o` (для компиляторов в Си вроде KCL [YH85], Scheme→C [Bar89] или Bigloo [Ser94]) или какой-нибудь `*.fasl`-файл для других компиляторов. Как бы то ни было, в этом файле останется лишь откомпилированный код раскрытых выражений, то есть определения сопутствующих функций. Собственно класс создаётся только в памяти компилятора во время раскрытия макросов, но не в памяти той лисп-системы, к которой будет подключен `*.o`-файл или которая динамически загрузит `*.fasl`-образ. Классы просто испаряются после того, как компилятор закончит свою работу.

Так что если мы хотим иметь поддержку рефлексии, то объекты-классы должны создаваться в мире программ. Однако, если и сами классы будут определяться тогда же, то их сопутствующие функции нельзя будет генерировать, потому как этот процесс должен выполняться во время макрораскрытия,[8] но ему необходим перечень всех полей класса. Следовательно, класс должен существовать в обоих мирах: и при раскрытии макросов, и после него. В качестве приятного дополнения, такой подход заодно позволит определять подклассы вместе с их суперклассами в одном общем файле. В случае, когда мы имеем дело с единым миром [**см. стр. 372**], MEROONET может попытаться создать класс дважды: первый раз при раскрытии макросов, а второй — при исполнении раскрытого кода. Чтобы избежать такого поведения,[9] применяется следующий хитрый трюк: пусть определена глобальная переменная `*last-defined-class*`, тогда если во время раскрытия макросов в неё поместить ссылку на первый попавшийся класс, то в едином мире это значение перейдёт в мир программ, а во множественных мирах переменная сохранит исходное значение. Это позволит узнать, когда определяемый класс не требуется создавать повторно.

```
(define *last-defined-class* #f)

(define-meroonet-macro (define-class name super-name own-fields)
  (set! *last-defined-class* #f)
  (let ((class (register-class name super-name own-fields)))
    (set! *last-defined-class* class)
    `(begin
       (if (not *last-defined-class*)
         (register-class ',name ',super-name ',own-fields) )
       ,(Class-generate-related-names class) ) ) )
```

Прочие затруднения

Продолжая иллюстрировать трудности использования макросов в реальных проектах (на примере MEROONET), стоит ещё раз задуматься о важности порядка раскрытия для макросов со внутренним состоянием наподобие `define-class`. Рассмотрим следующую форму:

```
(begin (define-class Point Object (x y))
       (define-class Polygon Point ((* side))) )
```

Если раскрытие производится слева направо, то класс `Point` определяется до класса `Polygon` и всё замечательно. Если же порядок будет противоположным, то класс `Polygon` определить не получится, не имея на руках его

[8] Вспомните, что в Scheme нельзя ни создавать, ни обращаться к переменным с динамически вычисляемыми именами. Все имена всех используемых переменных после раскрытия макросов должны быть записаны прямым текстом.

[9] MEROONET не придаёт чёткого смысла переопределению классов, поэтому не стоит лишний раз искушать судьбу.

суперкласса. Эту проблему, в принципе, можно обойти, если отложить создание `Polygon` до момента раскрытия определения `Point`. Тогда определение `Polygon` фактически станет пустым, а форма, создающая `Point`, должна будет раскрыться в определения обоих классов в правильном порядке.

А что если `Point` и `Polygon` определяются в одном файле, а класс `Colored-Polygon` — в другом? Теперь-то уже никак не отвертеться, все поля суперкласса надо каким-то образом извлечь и передать между файлами. MEROONET не забивает себе голову подобными трудностями и просто требует, чтобы все зависимые друг от друга классы компилировались вместе. MEROON же применяет ещё одну хитрость. Каждый родительский класс должен быть или ранее определён в этом же файле, или помечен ключевым словом `:prototype`. Такая форма лишь[10] помещает соответствующий класс в иерархию наследования, но не генерирует никаких функций. Выглядит это вот так:

```
(define-class Polygon Point ((* side)) :prototype)
(define-class ColoredPolygon Polygon (color))
```

Наиболее правильным решением, пожалуй, будет поддержка модулей с механизмом импорта/экспорта, позволяющим указать, что надо каждому модулю для компиляции и откуда это брать. В принципе это сводится к тому, что внутреннее состояние компилятора выносится в некую базу данных, которая используется для разрешения неудовлетворённых зависимостей между определениями.

11.4. Представление классов

Классы MEROONET представляются объектами MEROONET. Такой подход значительно облегчает реализацию рефлексивных возможностей объектной системы. Метаметоды также легче писать, оперируя структурой классов напрямую, а не косвенно опираясь на наследование. Методы, переносящие объекты с одного компьютера на другой, или, например, универсальные методы вывода объектов на экран очевидно удобнее определять, имея полное представление о структуре обрабатываемых объектов, об устройстве их классов. Прародителем всех объектов является класс `Object` — корень иерархии наследования. Все классы сами являются экземплярами класса `Class`. Поля представляются объектами классов `Mono-Field` и `Poly-Field`, наследников `Field`.

В процессе раскрытия определений классов используется множество вспомогательных функций, работающих с предопределёнными классами: `Mono-Field?`, `make-Poly-Field` и т. д. Значит ли это, что сопутствующие функции класса становятся доступными после его определения? Например, сможем ли мы использовать в определениях других классов функцию `make-Point` после

[10] На самом деле в MEROON всё немного сложнее: модификатор `:prototype` раскрывается в проверку того, что к моменту исполнения программы реальный класс был создан и написанный прототип ему соответствует.

того, как определим класс `Point`? Для текущей версии `define-class` ответ на этот вопрос отрицательный, так как `make-Point` создаётся не в процессе раскрытия, а уже во время исполнения раскрытого кода. Такой подход несколько усложняет реализацию метаклассов при компиляции, но MEROONET обходит стороной эту проблему.

Стараясь не перегружать классы, мы помещаем в них лишь самое необходимое: имя, числовой идентификатор, список полей, ссылку на суперкласс, список идентификаторов подклассов. Этот перечень дочерних классов пригодится для реализации обобщённых функций. Список хранит номера вместо прямых ссылок с целью избежать зацикливания. Итого, `Class` — класс всех классов — определяется следующим образом:

```
(define-class Class Object
  ( name number fields superclass subclass-numbers ) )
```

Поля классов, естественно, тоже являются объектами MEROONET. Поле характеризуется своим видом, именем и классом, которому оно принадлежит. Здесь ссылка на класс тоже сделана числом, чтобы поля можно было нормально распечатывать. Таким образом, имеем:

```
(define-class Field Object (name defining-class-number))
(define-class Mono-Field Field ())
(define-class Poly-Field Field ())
```

Наконец, иногда может потребоваться хозяин поля собственной персоной, а не только его номер. Мы всегда очень вежливы и точно знаем, кто нам нужен, поэтому следующая функция не утруждает себя излишними проверками:

```
(define (Field-defining-class field)
  (number->class (careless-Field-defining-class-number field)) )
```

Конечно, на самом деле все эти классы не получится определить до того, как MEROONET будет загружена, но и загрузить MEROONET без них тоже не выйдет. Чтобы разорвать данный порочный круг, придётся определить их вручную:

```
(define Object-class
  (vector
   1                          ; это класс
   'Object                    ; имя
   0                          ; номер
   '()                        ; поля
   #f                         ; нет суперкласса
   '(1 2 3)                   ; дочерние классы
   ) )

(define Class-class
  (vector
   1                                ; это тоже класс
```

```
   'Class                              ; имя
   1                                   ; номер
   (list                               ; поля
    (vector 4 'name             1)     ; смещение 1
    (vector 4 'number           1)     ; смещение 2
    (vector 4 'fields           1)     ; смещение 3
    (vector 4 'superclass       1)     ; смещение 4
    (vector 4 'subclass-numbers 1) )   ; смещение 5
   Object-class                        ; суперкласс
   '()
   ) )

(define Generic-class
  (vector
   1
   'Generic
   2
   (list
    (vector 4 'name           2)
    (vector 4 'default        2)
    (vector 4 'dispatch-table 2)
    (vector 4 'signature      2) )
    Object-class
    '()
    ) )

(define Field-class
  (vector
   1
   'Field
   3
   (list
    (vector 4 'name                  3)
    (vector 4 'defining-class-number 3) )
   Object-class
   '(4 5)
   ) )

(define Mono-Field-class
  (vector 1
          'Mono-Field
          4
          (careless-Class-fields Field-class)
          Field-Class
          '() ) )
```

```
(define Poly-Field-class
  (vector 1
          'Poly-Field
          5
          (careless-Class-fields Field-class)
          Field-Class
          '() ) )
```

Затем классы ставятся на положенные места:

```
(vector-set! *classes* 0 Object-class)
(vector-set! *classes* 1 Class-class)
(vector-set! *classes* 2 Generic-class)
(vector-set! *classes* 3 Field-class)
(vector-set! *classes* 4 Mono-Field-class)
(vector-set! *classes* 5 Poly-Field-class)

(set! *class-number* 6)
```

Так как MEROONET основана на иерархии классов, которую сама же определяет, некоторые функции (вроде аксессоров `Class-number` и `Class-fields`) требуются ещё до того, как они будут созданы. С этой целью мы определим их эквиваленты, начинающиеся на `careless-`. Имена говорят сами за себя: данные функции никак не проверяют свои аргументы; это приемлемо, так как они используются лишь внутри MEROONET, где всё находится под контролем.

```
(define (careless-Class-name class)
  (vector-ref class 1) )
(define (careless-Class-number class)
  (vector-ref class 2) )
(define (careless-Class-fields class)
  (vector-ref class 3) )
(define (careless-Class-superclass class)
  (vector-ref class 4) )

(define (careless-Field-name field)
  (vector-ref field 1) )
(define (careless-Field-defining-class-number field)
  (vector-ref field 2) )
```

11.5. Сопутствующие функции

Формирование имён этих функций значительно облегчит следующее вспомогательное определение:

```
(define (symbol-concatenate . names)
  (string->symbol (apply string-append (map symbol->string names))) )
```

Навскидку можно придумать два варианта конструирования функций, сопутствующих определениям классов. В первом случае их код помещается непосредственно в соответствующие определения; во втором же эти определения сводятся к формам, генерирующим необходимые замыкания во время исполнения. Второй вариант позволяет отделить неизменяемую часть функций, что облегчает их понимание и реализацию, но уменьшает эффективность, так как вызываемые функции не будут фиксированы на этапе компиляции, что усложняет оптимизацию. Различия двух подходов отлично видны на примере аллокатора класса `Polygon`:

```
(define allocate-Polygon
        (lambda (size)
          (let ((o (make-vector (+ 1 2 1 size))))
            (vector-set! o 0 (careless-Class-number Polygon-class))
            (vector-set! o 3 size)
            o ) ) )

(define allocate-Polygon (make-allocator Polygon-class))
```

В первом случае компилятору статически известно, что аллокатор является унарной функцией, а его тело состоит из вызовов тривиальных библиотечных функций вроде аксессоров векторов. Следовательно, это прекрасный кандидат на инлайнинг и его, например, можно сразу компилировать в виде макроса Си. Но вот второе определение ничего не говорит об арности. Фактически, нельзя даже с уверенностью сказать,[11] будет ли с переменной `allocate-Polygon` связана функция или что-то другое. Но несмотря на это, мы всё равно изберём второй путь, так как первый сложнее устроен и его описание более запутанно, а кроме того, он требует существенно больше памяти как при компиляции, так и в результирующем коде. Итак, сопутствующие функции будут генерироваться вот так:

```
(define (Class-generate-related-names class)
  (let* ((name (Class-name class))
         (class-variable-name (symbol-concatenate name '-class))
         (predicate-name      (symbol-concatenate name '?))
         (maker-name          (symbol-concatenate 'make- name))
         (allocator-name      (symbol-concatenate 'allocate- name)) )
    `(begin
       (define ,class-variable-name (->Class ',name))
       (define ,predicate-name (make-predicate ,class-variable-name))
       (define ,maker-name     (make-maker     ,class-variable-name))
       (define ,allocator-name (make-allocator ,class-variable-name))
       ,@(map (lambda (field) (Field-generate-related-names field class))
              (Class-fields class) )
       ',(Class-name class) ) ) )
```

[11] Лисп вообще не гарантирует, что вычисление этой формы будет завершено, а в Scheme она вполне может вернуть несколько значений вместо предполагаемого одного.

11.5.1. Предикаты

Для каждого класса MEROONET определяется предикат, способный отличать объекты данного класса и его наследников от всех остальных. Необычайно важно, чтобы этот предикат работал с максимально возможной скоростью, так как Scheme — это язык с динамической типизацией, где типы постоянно проверяются и перепроверяются. При компиляции иногда можно предсказать[12] тип объекта, объединить некоторые проверки, извлечь необходимую информацию из пользовательских определений или даже потребовать полной типобезопасности программ, например, предоставив функции, возвращающие типы своих аргументов.

Проверка принадлежности объекта классу является фундаментальной операцией языка, выполняемой предикатом `is-a?`. Ради быстродействия `is-a?` предполагает, что его аргумент действительно является объектом, а класс — классом. Следовательно, нельзя свободно применять `is-a?` к чему попало. В противоположность ему, предикаты, связанные с конкретными классами, более устойчивы: они сразу отбраковывают аргументы, который не похожи на объекты MEROONET. Наконец, полезным будет ещё один, самый строгий предикат, убеждающийся в принадлежности объекта указанному классу и выводящий понятное сообщение об ошибке в случае несовпадения. Так как в Scheme нет[13] стандартного механизма обработки ошибок, то MEROONET сообщает о них, вызывая функцию `meroonet-error`, которая... не определена — гарантированно переносимый способ получить ошибку!

```
(define (make-predicate class)
  (lambda (o) (and (Object? o)
                   (is-a? o class) )) )

(define (is-a? o class)
  (let up ((c (object->class o)))
    (or (eq? class c)
        (let ((sc (careless-Class-superclass c)))
          (and sc (up sc)) ) ) ) )

(define (check-class-membership o class)
  (if (not (is-a? o class))
    (meroonet-error "Wrong class" o class)
    #t ) )
```

Предикат `is-a?` имеет вычислительную сложность $O(n)$, так как сначала проверяется класс объекта, затем его суперкласс, затем суперкласс суперкласса и так далее. Но это в худшем случае; чаще всего достаточно одной проверки, реже — двух-трёх.

[12] Например, форма `define-method` явно указывает тип дискриминирующего аргумента.

[13] По крайней мере, в стандарте R^{5}RS, который взят за основу в этой книге.

Стоит отметить, что в формулировке предиката `is-a?` заботливо оставлены грабли в виде возможной бесконечной регрессии. Мы их успешно обошли, использовав функцию `careless-Class-superclass` для получения суперкласса вместо кажущейся более логичной `Class-superclass`. Разница между ними в том, что первая функция предполагает, что её аргумент является классом. В контексте `is-a?` мы не особо рискуем, делая подобные предположения. Но вот вторая функция обязана быть более аккуратной, и она наверняка проверяет, что её аргумент — это действительно класс. Естественно, это приведёт к рекурсивному вызову `is-a?`, что самым печальным образом скажется на производительности MEROONET.

11.5.2. Аллокатор без инициализации

MEROONET предоставляет два вида аллокаторов. Первый, называемый просто аллокатором, лишь выделяет в памяти место под объекты, не утруждая себя его инициализацией. Второй, именуемый конструктором, создаёт объекты с чётко определёнными значениями полей. Аналогичные понятия есть и в Scheme: `cons` — это конструктор точечных пар; `vector` создаёт и инициализирует векторы. Существует также и другой способ получить новый вектор — функция `make-vector`, если ей передать только один аргумент, возвращает вектор указанного размера с неопределённым содержимым. MEROONET поддерживает обе разновидности.

Аллокаторы создают объекты с неопределённым начальным состоянием. Существует как минимум два варианта понимания неопределённости. По причинам, связанным со сборкой мусора (если сборщики не пользуются некоторыми хитростями, см. [BW88]), неопределённое состояние часто понимается как неизвестное для пользователя языка, но вполне определённое для реализации. У такого подхода есть два подварианта, различающиеся семантикой используемых значений.

- Неопределённые поля могут быть помечены специальной сущностью `#<uninitialized>`, которая показывает отсутствие значения. Сама она не является значением в строгом смысле, поэтому попытка прочитать содержимое такого поля приводит к ошибке.

- Если реализация не поддерживает подобные сущности, то она может записывать в поле произвольное нормальное значение. В Лиспе, например, это зачастую `nil`, в Le_Lisp — `t`, многие реализации Scheme используют `#f`. Следовательно, пользователь не может полагаться на какие-либо конкретные начальные значения неинициализированных полей, но обращения к ним не будут принципиальной ошибкой.

И есть ещё третья интерпретация — известная как «C-style», — по которой обращение к неинициализированному полю приводит к неопределённым последствиям. Поэтому от греха подальше лучше вообще никогда не пытаться

читать неинициализированные поля. Такое понимание неинициализированности очень удобно для разработчиков языка, но отнюдь не для его пользователей, потому как термин «неопределённое поведение» может означать действительно *что угодно*. Возьмём, например, какой-нибудь аксессор чтения. Очевидно, его быстродействие должно быть максимально высоким, поэтому сам он принципиально не проверяет, что и откуда там считывается. А так как реализация ничего не гарантирует в случае неинициализированных полей, то в зависимости от настроения и текущей фазы Луны она может как выбросить мерзкую, но кристально понятную ошибку `"Bus error, core dumped"`, так и молча выдать какой-нибудь мусор, который программа примет за чистую монету. Вся ответственность за чтение неинициализированных полей возлагается на пользователей — справедливо, но очень опасно.

Первый подход (с `#<uninitialized>`) требует, чтобы обращение к полю, которое не было инициализировано, вызывало ошибку. Очевидно, это условие необходимо проверять всякий раз, когда производится обращение к потенциально неинициализированному полю [Que93b]. Второй вариант, равно как и третий, не нуждается в лишних проверках.

Сишный вариант, конечно, является наиболее эффективным с точки зрения скорости, так как выделенную память не надо ни инициализировать, ни очищать. Если собрать все эти утверждения воедино, то станет очевидным парадоксальный факт: в итоге изначально инициализированные объекты оказываются эффективнее, нежели объекты без инициализации. Это для пользователя начальные значения полей не имеют смысла, но реализация обязана помещать туда конкретные значения вроде `#<uninitialized>`; с таким же успехом она могла бы заполнять их чем-то более осмысленным. Наконец, чаще всего поля, не инициализированные при создании объекта, всё равно инициализируются впоследствии — а это двойная работа. В некоторых случаях её можно простить, но когда все значения известны заранее, глупо не использовать явную инициализацию, формирующую объект за один проход вместо двух.

CLOS понимает неопределённость в первом смысле и гарантирует обнаружение ошибок доступа к неинициализированным полям. В Scheme есть формы, для реализации которых может потребоваться понятие неинициализированных переменных. [**см. стр. 83**] По нашей задумке, MEROONET должна быть способна полностью эмулировать Scheme, а значит, иметь какой-то механизм представления таких переменных. Но дабы не усложнять реализацию, мы будем просто заполнять неинициализированные поля значением `#f`.

Концепция аллокации без инициализации имеет смысл только для изменяемых объектов. Это очень важное замечание, так как неизменяемые значения проще определить математически, чем объекты, имеющие состояние, что помогает при их сравнении. [**см. стр. 155**] Кроме того, неизменяемые объекты гораздо лучше оптимизируются в силу того, что все их поля неизменны. Возьмём, например, функцию `make-allocator`, которая определена чуть ниже. В ней используется форма `(Class-number class)`, значение которой вполне

можно было бы вычислить заранее и подставить напрямую (как это сделано для `(Class-fields class)`), если бы номер класса был неизменной величиной. Но из-за возможности переименования классов мы вынуждены выяснять его каждый раз заново.

Функция `make-allocator` создаёт аллокаторы. Она берёт класс и возвращает функцию, принимающую список натуральных чисел, соответствующих длине каждого индексированного поля этого класса. Полученная функция сначала определяет размеры необходимого участка памяти (длину вектора), складывая длины всех полей. Откуда она про них узнает? Их список извлекается из класса методом `Class-fields`. После того, как требуемый кусок памяти наконец получен, ему необходимо придать форму, записав в соответствующие места длины всех индексированных полей. Для этого организуется второй цикл, проходящий по всем полям класса и отслеживающий текущее смещение в выделенной области памяти. Наконец, получившийся «скелет» объекта связывается со своим классом и возвращается из аллокатора.

```
(define (make-allocator class)
  (let ((fields (Class-fields class)))
    (lambda sizes
      ;; вычисляем размер создаваемого объекта
      (let ((room (let iter ((fields fields)
                             (sizes sizes)
                             (room *starting-offset*) )
                    (if (pair? fields)
                        (cond ((Mono-Field? (car fields))
                               (iter (cdr fields) sizes (+ 1 room)) )
                              ((Poly-Field? (car fields))
                               (iter (cdr fields) (cdr sizes)
                                     (+ 1 (car sizes) room) ) ) )
                        room ) )))
        (let ((o (make-vector room #f)))
          ;; формируем родственные связи объекта и его скелет
          (vector-set! o 0 (Class-number class))
          (let iter ((fields fields)
                     (sizes sizes)
                     (offset *starting-offset*) )
            (if (pair? fields)
                (cond ((Mono-Field? (car fields))
                       (iter (cdr fields) sizes (+ 1 offset)) )
                      ((Poly-Field? (car fields))
                       (vector-set! o offset (car sizes))
                       (iter (cdr fields) (cdr sizes)
                             (+ 1 (car sizes) offset) ) ) )
                o ) ) ) ) ) ) )
```

Следует сделать несколько замечаний касательного этого кода:

1) Аллокаторы единственным аргументом принимают список длин полей. Следовательно, они неявно требуют создания в памяти этого списка, состоящего из точечных пар, — чтобы выделить память, вначале требуется выделить память! Позже мы рассмотрим способ этого избежать.

2) Лишние элементы данного списка просто игнорируются, не вызывая никаких ошибок.

3) Аллокатор дважды проходит по одному и тому же списку полей класса. Это не сильно эффективно, особенно для классов без индексированных полей, чей размер неизменен. Этот недочёт впоследствии тоже будет исправлен.

4) Если не дай бог указанный размер индексированного поля будет отрицательным, то необходимый объём памяти будет вычислен неверно и пользователь рано или поздно получит какое-то малопонятное сообщение об ошибке. Это не очень хорошо, поэтому следует выполнять явную проверку и заранее выводить своё, вменяемое сообщение.

5) Поля могут быть только экземплярами классов `Mono-Field` или `Poly-Field`. Пользовательские классы полей при такой реализации добавить невозможно.

11.5.3. Аллокатор с инициализацией

Конструкторы классов определяются похожим образом, но мы немного оптимизируем создание объектов небольших размеров. В Scheme аллокаторы (вроде `make-vector` или `make-string`) принимают размер создаваемого объекта, а также необязательное значение-заполнитель, которым инициализируется по умолчанию его содержимое. Конструкторы (наподобие `cons`, `vector` или `string`) принимают соответствующее количество значений для всех полей создаваемых объектов. Так как в векторе или строке только одно индексированное поле, то его длина без проблем определяется по количеству аргументов, переданных конструктору. У нас же таких полей может быть несколько, так что если не вводить для их инициализации особый синтаксис, то соответствующие длины необходимо указывать явно. В Meroonet они записываются перед перечислением содержимого подобных полей. Итого, цветной многоугольник с тремя сторонами создаётся следующим образом:

```
(make-ColoredPolygon 'x 'y 3 'Point0 'Point1 'Point2 'color)
```

Для аллокаторов с инициализацией мы применим следующий подход: все аргументы собираются в один общий список `params`; затем создаётся вектор, в который укладываются элементы этого списка вместе с номером класса. После этого остаётся только проверить, что структура созданного объекта

корректна с точки зрения MEROONET. Для этого мы пройдёмся по всем аргументам и полям класса, убеждаясь в том, что все поля инициализированы. Если всё хорошо, то сконструированный объект возвращается из функции. Такой подход проще (и, вероятно, быстрее), чем кажущийся более естественным предыдущий, где аргументы проверяются до того, как объект создаётся в памяти.

```
(define (make-maker class)
  (or (make-fix-maker class)
      (let ((fields (Class-fields class)))
        (lambda params
          ;; создаём объект
          (let ((o (apply vector (Class-number class) params)))
            ;; проверяем его скелет
            (let check ((fields fields)
                        (params params)
                        (offset *starting-offset*) )
              (if (pair? fields)
                  (cond ((Mono-Field? (car fields))
                         (check (cdr fields) (cdr params) (+ 1 offset)) )
                        ((Poly-Field? (car fields))
                         (check (cdr fields)
                                (list-tail (cdr params) (car params))
                                (+ 1 (car params) offset) ) ) )
                  o ) ) ) ) ) ) )
```

Этот аллокатор тоже игнорирует лишние аргументы, но он всё ещё не особо эффективен, так как аргументы собираются в список, который тут же превращается в вектор. Далее, из-за того, что для проверки структуры создаваемого объекта используется список `params`, а не полученный вектор, будет весьма непросто убедить компилятор в том, что этот список бесполезен и его можно не создавать. Поэтому мы прикрутим к `make-maker` небольшой костыль, чтобы исправить ситуацию хотя бы для объектов, не имеющих индексированных полей.

```
(define (make-fix-maker class)
  (define (static-size? fields)
    (if (pair? fields)
        (and (Mono-Field? (car fields))
             (static-size? (cdr fields)) )
        #t ) )
  (let ((fields (Class-fields class)))
    (and (static-size? fields)
         (let ((size (length fields))
               (cn (Class-number class)) )
           (case size
             ((0)  (lambda ()              (vector cn)))
             ((1)  (lambda (a)             (vector cn a)))
```

```
((2)  (lambda (a b)           (vector cn a b)))
((3)  (lambda (a b c)         (vector cn a b c)))
((4)  (lambda (a b c d)       (vector cn a b c d)))
((5)  (lambda (a b c d e)     (vector cn a b c d e)))
((6)  (lambda (a b c d e f)   (vector cn a b c d e f)))
((7)  (lambda (a b c d e f g) (vector cn a b c d e f g)))
(else #f) ) ) ) ) )
```

Таким образом, конструкторы классов, имеющих менее девяти неиндексированных полей, будут эффективными функциями с фиксированной арностью. Если класс не содержит индексированных полей, но в нём слишком много обычных, то для него тоже применяется стратегия со списком и явной проверкой. Кстати, если вы заметили, для индексированных полей эта проверка выполняется весьма элегантным образом с помощью `list-tail` и `cdr`: если в списке окажется недостаточно элементов, то одна из этих функций выдаст ошибку, так как они обе неприменимы к слишком коротким (в том числе пустым) спискам. Для небольших же объектов эта проверка тождественна обычной проверке арности. Естественно, подобное разнообразие стратегий отлова одной и той же ошибки означает нарушение единообразия сообщений о ней, что не очень красиво выглядит, но повсеместная реализация данных удобств раздула бы исходный код MEROONET более чем на четверть.

Действительно эффективно создавать объекты MEROONET можно только при непосредственной поддержке этой системы самим языком. Нелепостей вроде функции выделения памяти, требующей для работы сравнимого объёма памяти, можно избежать, например, имея в распоряжении внутреннюю функцию, создающую аллокаторы вида `(vector cn a b c ...)`.

11.5.4. Аксессоры

Для каждого поля класса в MEROONET определяются вспомогательные функции, позволяющие читать, писать и узнавать длину этого поля, если оно индексировано. Определения данных функций генерируются подфункцией `Field-generate-related-names`:

```
(define (Field-generate-related-names field class)
  (let* ((fname (careless-Field-name field))
         (cname (Class-name class))
         (cname-variable (symbol-concatenate cname '-class))
         (reader-name (symbol-concatenate cname '- fname))
         (writer-name (symbol-concatenate 'set- reader-name '!)) )
    `(begin
       (define ,reader-name
         (make-reader
          (retrieve-named-field ,cname-variable ',fname) ) )
       (define ,writer-name
         (make-writer
          (retrieve-named-field ,cname-variable ',fname) ) )
```

```
      ,@(if (Poly-Field? field)
            `((define ,(symbol-concatenate reader-name '-length)
                (make-lengther
                 (retrieve-named-field ,cname-variable ',fname) ) ))
            '() ) ) ) )
```

Как и для всех остальных сопутствующих классам функций, код аксессоров извлекается из соответствующих замыканий, а не генерируется напрямую. Конструкторы `make-reader`, `make-writer` и `make-lengther` принимают поле и возвращают необходимые замыкания. Так как аксессоры создаются динамически, данные функции вынуждены обращаться к глобальной переменной, содержащей информацию о классе, и искать там нужное поле по имени; для этого используется функция `retrieve-named-field`:

```
(define (retrieve-named-field class name)
  (let search ((fields (careless-Class-fields class)))
    (and (pair? fields)
         (if (eq? name (careless-Field-name (car fields)))
             (car fields)
             (search (cdr fields)) ) ) ) )
```

11.5.5. Чтение полей

Так как поля бывают двух видов: индексированные и обычные, то и аксессоры должны производиться в двух вариантах, различающихся арностью. Также сейчас самое время заметить, что смещения полей, расположенных до конца первого индексированного поля, не зависят от определения класса. Функция `make-reader`, создающая аксессоры чтения, конечно же, не должна забывать о данной возможности оптимизации. Мы воспользуемся списком полей класса для проверки неизменности смещения обрабатываемого поля. Если это так, то `make-reader` сгенерирует соответствующий специализированный аксессор; в противном случае будет использована стандартная функция, построенная на основе универсального аксессора `field-value`. Таким образом, все поля, имеющие фиксированные смещения, получат более эффективные аксессоры.

Аксессоры типобезопасны — они проверяют, соответствует ли класс передаваемых им объектов родительскому классу поля, с которым связан конкретный аксессор. За это отвечает функция `check-class-membership`, рассмотренная ранее. Более того, аксессоры индексированных полей должны проверять запрашиваемые индексы на соответствие размерам этих полей. С этим им поможет функция `check-index-range`.

```
(define (make-reader field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
```

```
        (if (eq? field (car fields))
            (cond ((Mono-Field? (car fields))
                   (lambda (o)
                     (check-class-membership o class)
                     (vector-ref o offset) ) )
                  ((Poly-Field? (car fields))
                   (lambda (o i)
                     (check-class-membership o class)
                     (check-index-range i o offset)
                     (vector-ref o (+ offset 1 i)) ) ) )
            (cond ((Mono-Field? (car fields))
                   (skip (cdr fields) (+ 1 offset)) )
                  ((Poly-Field? (car fields))
                   (cond ((Mono-Field? field)
                          (lambda (o)
                            (field-value o field) ) )
                         ((Poly-Field? field)
                          (lambda (o i)
                            (field-value o field i) ) ) ) ) ) ) ) ) )
```

```
(define (check-index-range i o offset)
  (let ((size (vector-ref o offset)))
    (if (not (and (<= 0 i) (< i size)))
        (meroonet-error "Index out of range" i size)
        #t ) ) )
```

Для доступа к остальным полям, находящимся за первым индексированным, применяется функция `field-value`. Она является обобщённой в смысле универсальности — позволяет получить доступ к любому полю любого класса. Реализуется она всё же как нормальная, не обобщённая функция, потому что MEROONET всё равно не позволяет пользователям определять свои классы полей. Естественно, у функции `field-value` есть сестра-близнец `set-field-value!`. Им обеим необходимо уметь динамически вычислять смещения полей в произвольном объекте, так что соответствующий код вынесен в отдельную функцию `compute-field-offset`.

```
(define (compute-field-offset o field)
  (let ((class (Field-defining-class field)))
    ;; (assume (check-class-membership o class))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields)) offset
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (skip (cdr fields)
                       (+ 1 offset (vector-ref o offset)) ) ) ) ) ) ) )
```

```
(define (field-value o field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
          (offset (compute-field-offset o field)) )
      (cond ((Mono-Field? field)
             (vector-ref o offset) )
            ((Poly-Field? field)
             (check-index-range (car i) o offset)
             (vector-ref o (+ offset 1 (car i))) ) ) ) ) )
```

11.5.6. Запись в поля

Определения аксессоров записи полностью аналогичны аксессорам чтения и не представляют существенных трудностей. Единственная функция, достойная внимания, — это `set-field-value!`, имеющая немного странную сигнатуру. Обычно порядок аргументов сеттера повторяет порядок в геттере, только в конце дописывается новое значение поля. Сравните, например, `car` и `set-car!`. Но в случае MEROONET необязательный индекс нарушает эту стройную систему, поэтому и был выбран порядок[14] `(o v field . i)`.

```
(define (make-writer field)
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (cond ((Mono-Field? (car fields))
                 (lambda (o v)
                   (check-class-membership o class)
                   (vector-set! o offset v) ) )
                ((Poly-Field? (car fields))
                 (lambda (o i v)
                   (check-class-membership o class)
                   (check-index-range i o offset)
                   (vector-set! o (+ offset 1 i) v) ) ) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (cond ((Mono-Field? field)
                        (lambda (o v)
                          (set-field-value! o v field) ) )
                       ((Poly-Field? field)
                        (lambda (o i v)
                          (set-field-value! o v field i) ) ) ) ) ) ) ) ) )
```

[14] Если вы ещё не забыли, что такое списки свойств, то наверяка вспомните и то, что аргументы `putprop` располагаются в точно таком же порядке.

```
(define (set-field-value! o v field . i)
  (let ((class (Field-defining-class field)))
    (check-class-membership o class)
    (let ((fields (careless-Class-fields class))
          (offset (compute-field-offset o field)) )
      (cond ((Mono-Field? field)
             (vector-set! o offset v) )
            ((Poly-Field? field)
             (check-index-range (car i) o offset)
             (vector-set! o (+ offset 1 (car i)) v) ) ) ) ) )
```

Немаловажным моментом является стиль именования сеттеров. Здесь используется префикс `set-`, подобно `set-cdr!`. В принципе, можно было бы использовать и суффикс `-set!`, как в `vector-set!`, но у префикса есть определённые преимущества: так сразу понятно, что данная функция модифицирует объекты. Да и читаются имена в этом случае естественнее.

11.5.7. Длина полей

Каждое индексированное поле получает особую функцию, позволяющую узнать его длину в конкретном объекте. Устроены эти функции так же, как и остальные аксессоры: если возможно, то генерируется оптимизированный вариант, иначе используется универсальная функция `field-length`.

```
(define (make-lengther field)
  ;; (assume (Poly-Field? field))
  (let ((class (Field-defining-class field)))
    (let skip ((fields (careless-Class-fields class))
               (offset *starting-offset*) )
      (if (eq? field (car fields))
          (lambda (o)
            (check-class-membership o class)
            (vector-ref o offset) )
          (cond ((Mono-Field? (car fields))
                 (skip (cdr fields) (+ 1 offset)) )
                ((Poly-Field? (car fields))
                 (lambda (o)
                   (field-length o field)) ) ) ) ) ) )

(define (field-length o field)
  (let* ((class  (Field-defining-class field))
         (fields (careless-Class-fields class))
         (offset (compute-field-offset o field)) )
    (check-class-membership o class)
    (vector-ref o offset) ) )
```

11.6. Создание классов

Форма `define-class` на самом деле не создаёт объект, представляющий определяемый класс. Она перепоручает эту работу функции `register-class`. Эта функция уже выделяет память и вызывает `Class-initialize!`, чтобы заполнить её осмысленными данными, полученными в результате анализа определения класса и его родословной. После формирования дескрипторов полей (с чем помогает функция `parse-fields`) класс помещается на подобающее ему место в иерархии наследования. Наконец, функция `update-generics` рассказывает всем обобщённым функциям о пополнении в семействе, обеспечивая корректное наследование методов.

```
(define (register-class name super-name own-fields)
  (Class-initialize! (allocate-Class)
                     name
                     (->Class super-name)
                     own-fields ) )

(define (Class-initialize! class name superclass own-fields)
  ;; инициализируем поля класса
  (set-Class-number!           class *class-number*)
  (set-Class-name!             class name)
  (set-Class-superclass!       class superclass)
  (set-Class-subclass-numbers! class '())
  (set-Class-fields!
   class (append (Class-fields superclass)
                 (parse-fields class own-fields) ) )
  ;; помещаем его в иерархию
  (set-Class-subclass-numbers!
   superclass
   (cons *class-number* (Class-subclass-numbers superclass)) )
  (vector-set! *classes* *class-number* class)
  (set! *class-number* (+ 1 *class-number*))
  ;; передаём в наследство методы родителей
  (update-generics class)
  class )
```

Описания полей в определениях классов разбираются функцией `parse-fields`. Она проводит синтаксический анализ полученного списка дескрипторов. Каждый дескриптор может быть или просто именем, или списком. Если это список, то он может начинаться лишь на знак равенства или звёздочку. Любая другая форма записи приводит к `meroonet-error`. MEROONET не позволяет переопределять наследуемые поля; функция `check-conflicting-name` гарантирует соблюдение данного запрета. С повторением имён собственных полей MEROONET поступает так же, как и с повторным определением классов: никак.

```
(define (parse-fields class own-fields)
  (define (Field-initialize! field name)
    (check-conflicting-name class name)
    (set-Field-name! field name)
    (set-Field-defining-class-number! field (Class-number class))
    field )

  (define (parse-Mono-Field name)
    (Field-initialize! (allocate-Mono-Field) name) )

  (define (parse-Poly-Field name)
    (Field-initialize! (allocate-Poly-Field) name) )

  (if (pair? own-fields)
      (cons (cond
             ((symbol? (car own-fields))
              (parse-Mono-Field (car own-fields)) )
             ((pair? (car own-fields))
              (case (caar own-fields)
                ((=) (parse-Mono-Field (cadr (car own-fields))))
                ((*) (parse-Poly-Field (cadr (car own-fields))))
                (else (meroonet-error
                       "Erroneous field description"
                       (car own-fields) )) ) ) )
            (parse-fields class (cdr own-fields)) )
      '() ) )

(define (check-conflicting-name class fname)
  (let check ((fields (careless-Class-fields (Class-superclass class))))
    (if (pair? fields)
        (if (eq? (careless-Field-name (car fields)) fname)
            (meroonet-error "Duplicated field name" fname)
            (check (cdr fields)) )
        #t ) ) )
```

11.7. Предопределённые сопутствующие функции

На данный момент мы описали практически всё, что необходимо для определения новых классов, за исключением одной маленькой детали. В самом деле, рассмотренные определения не заработают без функций, сопутствующих классам `Class`, `Field` и т. д. Форму `define-class` нельзя использовать, пока эти функции не будут определены, но проблема в том, что именно `define-class` и должна их определить. Опять мы оказались в порочном кругу.

Как обычно, разрывается такой круг волевым усилием, заключающемся в «ручном» определении всех необходимых функций. Не требуется определять их все, достаточно будет лишь минимально необходимого набора. Также можно не проводить каких-либо проверок вроде того, что `Class-class` — это действительно класс, и т. п. Если всё сделать правильно, то после этого можно будет передать `define-class` описания предопределённых классов и получить на выходе код, совпадающий с приведённым в начале главы с точностью до номеров классов. Единственное правило, которое требуется неукоснительно соблюдать: предикаты определяются первыми, геттеры — перед сеттерами, аллокаторы — в последнюю очередь. Ах да, и функции для класса `Class` должны быть определены перед аналогичными для остальных классов. Здесь приведена лишь часть необходимых определений, так как их полный список малоинтересен.

```
(define Class?   (make-predicate Class-class))
(define Generic? (make-predicate Generic-class))
(define Field?   (make-predicate Field-class))

(define Class-name
  (make-reader (retrieve-named-field Class-class 'name)))
(define set-Class-name!
  (make-writer (retrieve-named-field Class-class 'name)))
(define Class-number
  (make-reader (retrieve-named-field Class-class 'number)) )
(define set-Class-subclass-numbers!
  (make-writer (retrieve-named-field Class-class 'subclass-numbers)) )

(define make-Class (make-maker Class-class))
(define allocate-Class (make-allocator Class-class))

(define Generic-name
  (make-reader (retrieve-named-field Generic-class 'name)) )

(define allocate-Poly-Field (make-allocator Poly-Field-class))
```

Вот теперь `define-class` полностью готова к использованию. Осторожно: не следует тут же бежать «правильно» переопределять классы `Object`, `Class` и другие. Во-первых, из-за того, что `define-class` не идемпотентна,[15] вы получите шесть новых классов, а предыдущие шесть так и останутся бессмысленно болтаться в памяти. Во-вторых, компилятор не особо обрадует переопределение всех сопутствующих функций, до этого считавшихся неизменными.

[15] Правда, здесь можно схитрить и сбросить `*class-number*` в ноль перед переопределением встроенных классов. Только определять их надо в точно таком же порядке, как ранее вручную, чтобы номера остались неизменными.

11.8. Обобщённые функции

Обобщённые функции появились в результате адаптации идеи отправки сообщений (родом из объектно-ориентированных языков) к функциональному миру Лиспа. В Smalltalk [GR83] сообщения посылаются следующим образом:

получатель сообщение: *аргументы*...

Как известно, абсолютно любой язык можно превратить в Лисп, просто добавив в него побольше скобочек! Первые попытки перенести обмен сообщениями в Лисп использовали специальные ключевые слова наподобие `send` или `=>` (Planner [HS75]):

`(send` *получатель сообщение аргументы*...`)`

Получателями сообщений, естественно, являются объекты. Smalltalk позволяет отправить сообщение только одному конкретному объекту; в этом языке нельзя попросить несколько объектов совместно обработать какое-либо сообщение. Но даже в Лиспе легко увидеть огромное множество функций вроде банального сложения, чьи действия зависят от типа более чем одного объекта. И правда, целые числа складываются одним способом, дроби и комплексные числа — другим, у сложения чисел с плавающей запятой есть свои особенности. Если аргументы имеют различные типы, то их необходимо как-то свести к общему знаменателю; подходящий алгоритм приведения зависит от типов всех аргументов, а не только первого. Методы — это способы обработки конкретного сообщения конкретным объектом. Их обобщением на случай нескольких ответственных за обработку объектов являются *мультиметоды*. Однако, предыдущий синтаксис отправки сообщений ставит единственного получателя в привилегированное положение, поэтому Common Loops [BKK+86] был предложен более подходящий вариант:

`(`*сообщение объекты*...`)`

Ключевое слово `send` исчезло, а вместо него на место функции стало само сообщение — следовательно, это сообщение должно быть настоящей функцией. Точный смысл таких функций зависит от типов получателей сообщения, поэтому они называются *обобщёнными*. Обобщённые функции полагаются при выборе действий на свои аргументы, причём они обладают существенной свободой их трактовки:

1) Обобщённая функция сама определяет, какие аргументы влияют на её выбор. Такие аргументы называются *дискриминантами*.

2) Дискриминантом не обязательно является первый аргумент.

3) Дискриминантов может быть несколько, как в случае со сложением.

Короче говоря, обобщённые функции обобщают идею обмена сообщениями. Именно в таком виде эта идея реализована в MEROONET. С одной поправкой:

мы решили не включать в неё мультиметоды. Вспомните, ведь они ни разу не понадобились при объяснениях в этой книге. По мнению [KR90], мультиметоды действительно необходимы лишь в 5 % случаев. Это не настолько важная или невообразимо сложная для реализации функциональность, чтобы заострять здесь на ней особое внимание.

У вас вполне может возникнуть вопрос, применимы ли вообще подобные обобщения к объектам, ведь они являются «суверенными» сущностями, которые вполне способны самостоятельно разобраться, как им вызывать свои же методы. Мы не будем детально обсуждать данный вопрос в силу его философского характера. Достаточно отметить, что обобщённые функции полностью скрывают объектно-ориентированный аспект производимых ими действий: это просто функции, а не какой-то хитрый оператор `send`. Не важно, анализируют ли они типы аргументов или организуют обмен сообщениями между объектами — для пользователя они выглядят как функции, вызываются как функции и работают как функции. Хотя, естественно, с точки зрения реализации разница определённо есть. Рассмотрим обобщённую функцию g, чьим дискриминантом является первый аргумент. Каждый её вызов (`(`g *аргументы*`...)`) на самом деле может раскрываться в обращения к замыканию вида `(lambda args (apply send (car args)` g `(cdr args)))`, что однозначно сказывается на производительности.

Как же реализовать обобщённые функции на Scheme? С одной стороны, весьма желательно, чтобы для них работала `apply`, а это возможно лишь при условии, что обобщённые функции будут реализованы как функции Scheme. С другой стороны, не хотелось бы терять рефлексивные возможности, предоставляемые MEROONET, для чего необходимо оставить обобщённые функции объектами класса `Generic`. Итого, они должны стать одновременно объектами *и* функциями, то есть *функторами* — вызываемыми объектами. CLOS и Oaklisp [LP86, LP88] поддерживают эту концепцию непосредственно.

Без сомнения, обобщённые функции являются объектами, так как у них есть состояние — текущий набор поддерживаемых методов. Поэтому мы реализуем их следующим образом, пусть он и не особо оптимален. Каждая обобщённая функция представляется парой из объекта MEROONET и функции Scheme, чьё имя совпадает с именем обобщённой функции. Эта функция запомнит свой объект в замыкании, чтобы иметь возможность добраться до него в любом контексте. С другой стороны, данный объект будет доступен не только изнутри замыкания, что обеспечивает возможность добавления новых методов. Однако, для этого потребуется найти нужный объект обобщённой функции, для чего всем им потребуются некоторые идентификаторы (следовательно, обобщённые функции не могут быть анонимными). Глобальные переменные не используются потому, что это бы позволило локальным переменным (нечаянно) скрывать объекты, необходимые для корректной работы MEROONET.

Определения обобщённых функций имеют следующий синтаксис:

`(define-generic (`*имя аргументы*`...)` [*тело-по-умолчанию*]`)`

Первой указывается форма вызова функции и её дискриминант (получатель сообщения в терминах Smalltalk). Дискриминант записывается в скобках. Вся оставшаяся часть определения образует тело метода по умолчанию. В языке, где все значения являются объектами, это было бы аналогично определению метода для самого общего класса значений — `Object`. Но в нашем случае ещё остаются обычные значения Scheme, которые, очевидно, не являются объектами, поэтому метод по умолчанию необходимо также применять ко всем значениям, провалившим проверку `Object?`. Такой подход облегчает интеграцию MEROONET и Scheme, а также позволяет пользователю отлавливать ошибки типизации. Ещё одним достоинством MEROONET является поддержка всех возможных сигнатур обобщённых функций, включая точечные переменные. Естественно, методов это тоже касается.

Класс `Generic` определяется следующим образом:

```
(define-class Generic Object
  ( name default dispatch-table signature ) )
```

Поиском обобщённых функций по именам занимается функция `->Generic`. Поле `default` содержит метод по умолчанию, чьё тело или явно определяется пользователем, или содержит автоматически сгенерированную заглушку с сообщением об ошибке. Сохранённая сигнатура понадобится для проверки арности добавляемых методов на совместимость. Это очень важная проверка, так как, во-первых, вызовы обобщённых функций и так уже достаточно зависят от контекста — набора определённых методов, — добавление ещё одной степени свободы в виде арности только усложнит восприятие программ. Во-вторых, единообразие сигнатур методов позволяет эффективнее реализовывать вызовы обобщённых функций, см. [KR90].

Внутреннее состояние обобщённой функции состоит из единственного вектора, чьи индексы соответствуют номерам классов. Этот вектор хранит все определённые методы данной функции и называется *таблицей диспетчеризации* (dispatch table). Алгоритм вызова обобщённых функций теперь очевиден: номер класса дискриминанта — это индекс требуемого метода в таблице диспетчеризации обобщённой функции. Такой механизм выбора невероятно быстр, но эта скорость приобретается ценой повышенного потребления памяти: суммарно все такие векторы составляют матрицу $m \times n$, где m и n — это количество всех используемых программе классов и обобщённых функций соответственно. Однако, чаще всего в таких таблицах очень много пустых ячеек, что открывает возможности для оптимизации, см. например [VH94, Que93b].

Методы обобщённых функций (включая умолчательный) всегда имеют фиксированную арность, даже когда у функции есть точечная переменная. Например, если обобщённая функция `f` определена вот так:

```
(define-generic (f a (b) . c) (g b a c))
```

то её метод по умолчанию будет таким:

```
(lambda (a b c) (g b a c))
```

Остальные методы также должны иметь арность, совместимую с `(a b c)`. Функция, представляющая `f` в мире Scheme, получает сигнатуру, идентичную написанной в определении `f`:

```
(lambda (a b . c) ((determine-method G127 b) a b c))
```

Значение переменной `G127`[16] — это тот самый объект класса `Generic`, представляющий обобщённую функцию `f` со стороны MEROONET; в нём лежит таблица диспетчеризации, из которой `determine-method` достаёт соответствующие методы. Ввиду того, что не все значения Scheme являются объектами, перед поиском метода необходимо убедиться в том, что дискриминант — это действительно объект MEROONET.

```
(define (determine-method generic o)
  (if (Object? o)
      (vector-ref (Generic-dispatch-table generic)
                  (vector-ref o 0) )
      (Generic-default generic) ) )
```

Сами же обобщённые функции определяются следующим образом:

```
(define-meroonet-macro (define-generic call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((generic (gensym)))      ; соблюдаем гигиену
       `(define ,(car call)
          (let ((,generic (register-generic
                           ',(car call)
                           (lambda ,(flat-variables variables)
                             ,(if (pair? body)
                                  `(begin . ,body)
                                  `(meroonet-error
                                    "No method" ',(car call)
                                    . ,(flat-variables variables) ) ) )
                           ',(cdr call) )))
            (lambda ,variables
              ((determine-method ,generic ,(car discriminant))
               . ,(flat-variables variables) ) ) ) ) ) ) ) )
```

Функция `parse-variable-specifications` анализирует список аргументов, определяет в нём дискриминант и корректирует его синтаксис в соответствии с правилами Scheme. Эта функция используется как в `define-generic`,

[16] Так как `define-generic` — это макрос, то во избежание коллизий все имена используемых им переменных генерируются `gensym`.

так и в `define-method`. Полученные два значения она передаёт своему второму аргументу. Ради простоты функция `parse-variable-specifications` определена слегка небрежно: она не проверяет, единственен ли дискриминант.

```
(define (parse-variable-specifications specifications k)
  (if (pair? specifications)
      (parse-variable-specifications
       (cdr specifications)
       (lambda (discriminant variables)
         (if (pair? (car specifications))
             (k (car specifications)
                (cons (caar specifications) variables) )
             (k discriminant (cons (car specifications) variables)) ) ) )
      (k #f specifications) ) )
```

Вслед за списком аргументов конструируется тело обобщённой функции, после чего функция `register-generic` создаёт итоговый объект Meroonet. Эта функция позволяет скрыть (на уровне макроса `define-generic`) детали реализации обобщённых функций, в частности, список `*generics*`. Она же создаёт таблицу диспетчеризации, все ячейки которой изначально содержат метод по умолчанию — именно этот метод должен вызываться, если нет более подходящего. Размер таблицы диспетчеризации зависит от общего количества классов, а не текущего на момент создания обобщённой функции: при определении новых классов во всех таблицах должны добавляться новые ячейки. Так как у нас максимальное количество классов фиксировано, то достаточно будет просто зарезервировать вектор соответствующего размера.

```
(define (register-generic generic-name default signature)
  (let* ((dispatch-table (make-vector *maximal-number-of-classes*
                                      default ))
         (generic (make-Generic generic-name
                                default
                                dispatch-table
                                signature )) )
    (set! *generics* (cons generic *generics*))
    generic ) )
```

Функция `flat-variables` собирает все аргументы в простой список и преобразует точечную переменную в обычную. Это преобразование используется при определении всех методов.

```
(define (flat-variables variables)
  (if (pair? variables)
      (cons (car variables) (flat-variables (cdr variables)))
      (if (null? variables) variables (list variables)) ) )
```

Ещё немного об определениях классов

Как вы помните, при определении класса вызывается функция `update-generics`. Её задача заключается в распространении методов суперкласса на объекты создаваемого класса. Если для ранее определённого класса `Point` существует метод `show`, то логично будет сделать так, чтобы его подкласс `ColoredPoint` тоже обладал этим методом. Для этого потребуется обновить таблицы диспетчеризации всех обобщённых функций, которые были определены на момент создания нового класса.

```
(define (update-generics class)
  (let ((superclass (Class-superclass class)))
    (for-each
     (lambda (generic)
       (vector-set! (Generic-dispatch-table generic)
                    (Class-number class)
                    (vector-ref (Generic-dispatch-table generic)
                                (Class-number superclass) ) ) )
     *generics* ) ) )
```

11.9. Методы

Методы определяются с помощью формы `define-method`. Её синтаксис схож с `define`, только список аргументов метода выглядит так же, как и в `define-generic`: аргумент-дискриминант записывается в скобках вместе с именем класса, для которого определяется данный метод.

`(define-method (`*имя переменные*`...) `*формы*`...)`

Обобщённые функции являются объектами, чьё поведение можно динамически расширять, определяя новые методы с помощью `define-method`. Следовательно, обобщённые функции изменяемы, что несколько усложняет оптимизацию их вызовов, если только в языке нет возможности полностью или частично зафиксировать иерархию классов (подобно ключевому слову `sealed` в Dylan [App92b]), чтобы можно было спокойно провести её статический анализ. Другим решением будет сделать обобщённые функции неизменяемыми, но для этого необходимы функторы, которых в стандартном Scheme нет.

С арностью методов мы уже разобрались, когда рассматривали определение обобщённых функций и методов по умолчанию. Кроме этого надо не забыть распространить определяемый метод не только на непосредственно указанный класс, но также и на все его подклассы, для которых данный метод не был определён ранее.

Единственной оставшейся проблемой является реализация возможности, которая в Smalltalk называется `super`, а в CLOS — `call-next-method`. Имеется в виду возможность для метода подкласса вызвать одноимённый метод

суперкласса (который иначе для подкласса недоступен). Форма `(call-next-method)` может использоваться только внутри определений методов и означает вызов соответствующего суперметода с теми же[17] аргументами, что и у вызывающего метода. Естественно, суперметодами класса `Object` являются методы по умолчанию.

Функция `call-next-method` определяется локально в теле метода и реализуется аналогична рассмотренной ранее `determine-method`: нужный метод извлекается из таблицы диспетчеризации соответствующей обобщённой функции по номеру класса, просто в этот раз используется суперкласс рассматриваемого объекта. Внимание: по причинам, связанным с потенциальными особенностями макрораскрытия [**см. стр. 503**], невозможно гарантированно знать номера классов до начала исполнения программы. Поэтому мы поступим следующим образом. Форма `define-method` сначала определит предварительный метод, принимающий класс и обобщённую функцию; он должен будет вернуть замыкание, содержащее код настоящего метода. Затем, во время исполнения, этот предметод будет вызван с корректными аргументами, а полученное замыкание — установлено на его место. Всё это помещается в функцию `register-method`, чтобы скрыть детали реализации и не перегружать ими определение макроса `define-method`.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym)) (c (gensym)))
       `(register-method
         ',(car call)
         (lambda (,g ,c)
           (lambda ,(flat-variables variables)
             (define (call-next-method)
               ((if (Class-superclass ,c)
                    (vector-ref (Generic-dispatch-table ,g)
                                (Class-number (Class-superclass ,c)) )
                    (Generic-default ,g) )
                . ,(flat-variables variables) ) )
             . ,body ) )
         ',(cadr discriminant)
         ',(cdr call) ) ) ) ) )
```

Функция `register-method` определяет необходимый класс и обобщённую функцию, преобразует предметод в нормальный метод, проверяет согласованность его сигнатуры и сигнатуры обобщённой функции, после чего, наконец,

[17] В отличие от CLOS, мы запретили изменять аргументы вызова суперметода, так как иначе стало бы возможным заменить дискриминант абсолютно любым объектом, что противоречит самой идее суперметодов, а также мешает восприятию кода как людьми, так и компиляторами.

устанавливает метод в таблицу диспетчеризации. Проверку согласованности, в принципе, можно было бы выполнить ещё при раскрытии макросов, но для этого макрос `define-method` должен помнить все определённые на момент раскрытия обобщённые функции — короче говоря, мы бы получили те же сложности с состоянием, что и у `define-class`. Так как эта проверка весьма проста, выполняется всего лишь один раз и в дальнейшем никак не влияет на быстродействие обобщённых функций, то её с чистой совестью можно провести во время исполнения программы. Заметьте, что при обновлении методов функции сравниваются с помощью `eq?`.

```
(define (register-method generic-name pre-method class-name signature)
  (let* ((generic (->Generic generic-name))
         (class (->Class class-name))
         (new-method (pre-method generic class))
         (dispatch-table (Generic-dispatch-table generic))
         (old-method (vector-ref dispatch-table
                                 (Class-number class) )) )
    (check-signature-compatibility generic signature)
    (let propagate ((cn (Class-number class)))
      (let ((content (vector-ref dispatch-table cn)))
        (if (eq? content old-method)
            (begin
              (vector-set! dispatch-table cn new-method)
              (for-each propagate
               (Class-subclass-numbers (number->Class cn)) ) ) ) ) ) ) )

(define (check-signature-compatibility generic signature)
  (define (coherent-signatures? la lb)
    (if (pair? la)
        (if (pair? lb)
            (and (or
                  ;; совпадают дискриминанты
                  (and (pair? (car la)) (pair? (car lb)))
                  ;; совпадают аргументы
                  (and (symbol? (car la)) (symbol? (car lb))) )
              (coherent-signatures? (cdr la) (cdr lb)) )
            #f )
        (or (and (null? la) (null? lb))
            ;; совпадает точечная переменная
            (and (symbol? la) (symbol? lb)) ) ) )

  (if (not (coherent-signatures? signature
                                 (Generic-signature generic) ))
    (meroonet-error "Incompatible signature" generic signature)
    #t ) )
```

11.10. Заключение

В рассмотренных программах много чего можно улучшить как в плане повышения быстродействия, так и для улучшений рефлексивных качеств объектной системы. Ещё одной возможностью, достойной внимания, является встраивание MEROONET в реализацию языка, что открыло бы дорогу множеству других оптимизаций, а также сделало бы Scheme по-настоящему объектно-ориентированным. Именно из этих соображений EULISP, ILOG Talk, COMMON LISP и многие другие поддерживают концепцию объектов непосредственно, делая их базовыми понятиями языка.

11.11. Упражнения

Упражнение 11.1 Улучшите определение предиката `Object?`, чтобы он чётче различал объекты и значения Scheme.

Упражнение 11.2 Напишите обобщённую функцию `clone`, копирующую объекты MEROONET. Пусть она выполняет «поверхностное копирование» (shallow copy): содержимое полей исходного объекта можно просто перенести в копию без рекурсивных вызовов `clone`.

Упражнение 11.3 MEROONET позволяет определять новые типы классов — метаклассы, являющиеся наследниками `Class`. Реализуйте метакласс, чьими экземплярами будут классы, подсчитывающие количество своих объектов.

Упражнение 11.4 Классам и полям MEROONET не помешало бы чуть больше рефлексии. Добавьте им дополнительные поля, ссылающиеся на сопутствующие функции: предикат и оба аллокатора для классов; геттеры, сеттеры и измерители длины для полей.

Упражнение 11.5 CLOS не требует обязательного существования обобщённой функции для определения её метода. Доработайте `define-method` так, чтобы она создавала необходимую обобщённую функцию на лету, если она ещё не была определена.

Упражнение 11.6 Некоторые объектные системы, вроде CLOS и ΤΕΛΟΣ, наряду с `call-next-method` имеют также предикат `next-method?`, возвращающий ложь, когда следующий суперметод не определён. Эта рефлексивная возможность позволяет последовательно выполнить всю цепочку суперметодов, не рискуя при этом вызвать метод по умолчанию, потенциально приводящий к ошибке. Научите `define-method` определять функцию `next-method?`.

Рекомендуемая литература

Диалект T [AR88] предлагает иной подход к объектам в Scheme. Если вам понравилась идея метаобъектов, то сперва стоит взглянуть на вот эту статью о рефлексии в ObjVlisp: [Coi87], а после уже приниматься за изучение метаобъектного протокола CLOS [KdRB92].

Ответы к упражнениям

Упражнение 1.1

Просто поместите трассирующий код в соответствующее место — обработку вызовов функций:

```
(define (tracing.evaluate exp env)
  (if ...
      ...
      (case (car exp)
        ...
        (else (let ((fn   (evaluate (car e) env))
                    (args (evlis (cdr e) env)) )
                (display `(calling ,(car e) with . ,args)
                         *trace-port* )
                (let ((result (invoke fn args)))
                  (display `(returning from ,(car e) with ,result)
                           *trace-port* )
                  result ) )) ) ) )
```

Обратите внимание на два момента. Во-первых, здесь используется *имя* функции, а не её значение. Обычно сообщения в таком случае получаются понятнее. Во-вторых, `display` осуществляет вывод в настраиваемый `*trace-port*`. Это облегчает перенаправление результатов трассировки в специальное окно, или в лог-файл, или в стандартный поток вывода.

Упражнение 1.2

[Wan80b] приписывает изобретение этой оптимизации Дэниелу Фридмену и Дэвиду Уайзу. Она позволяет избавиться от бессмысленного вычисления выражения `(evlis '() env)` в конце последовательности. Кроме того, значение `env` неизменно в процессе вычисления элементов последовательности, поэтому его можно не включать лишний раз в список аргументов рекурсивно вызываемой функции. В среднем, `evlis` имеет дело со списками длиной порядка трёх-четырёх элементов, и у каждого из них обязательно есть конец, так что эта оптимизация оказывается весьма полезной. Кроме того, она экономит один вызов предиката.

```
(define (evlis exps env)
  (define (evlis exps)
    ;; (assume (pair? exps))
```

```
    (if (pair? (cdr exps))
        (cons (evaluate (car exps) env)
              (evlis (cdr exps)) )
        (list (evaluate (car exps) env)) ) )
  (if (pair? exps) (evlis exps) '()) )
```

Упражнение 1.3

Такое представление окружений известно как *гирлянда* ввиду очевидного визуального сходства. Здесь используется меньше точечных пар, но ценой этого является некоторое замедление поиска и модификации значений переменных. Более того, такая реализация `extend` делает невозможной проверку арности для функций с точечным аргументом.

```
(define (lookup id env)
  (if (pair? env)
      (let look ((names  (caar env))
                 (values (cdar env)) )
        (cond ((symbol? names)
               (if (eq? names id)
                   values
                   (lookup id (cdr env)) ) )
              ((null? names)
               (lookup id (cdr env)) )
              ((eq? (car names) id)
               (if (pair? values) (car values)
                   (wrong "Too few values") ) )
              (else (if (pair? values)
                        (look (cdr names) (cdr values))
                        (wrong "Too few values") )) ) )
      (wrong "No such binding" id) ) )
```

Реализация `update!` аналогична по структуре.

Упражнение 1.4

```
(define (s.make-function variables body env)
  (lambda (values current.env)
    (for-each (lambda (var val)
                (putprop var 'apval (cons val (getprop var 'apval))) )
              variables values )
    (let ((result (eprogn body current.env)))
      (for-each (lambda (var)
                  (putprop var 'apval (cdr (getprop var 'apval))) )
                variables )
      result ) ) )
```

```
(define (s.lookup id env)
  (car (getprop id 'apval)) )

(define (s.update! id env value)
  (set-car! (getprop id 'apval) value) )
```

Упражнение 1.5

Так как эта проблема касается не только `<`, то имеет смысл написать макрос, определяющий предикаты правильно. Обратите внимание, что для определяющего Лиспа значение `the-false-value` является истиной.

```
(define-syntax defpredicate
  (syntax-rules ()
    ((defpredicate name value arity)
     (defprimitive name
      (lambda values (or (apply value values) the-false-value))
      arity ) ) ) )

(defpredicate < < 2)
```

Упражнение 1.6

Главная сложность `list` в том, что это функция с переменной арностью. Однако, если вспомнить, как у нас представляются примитивы, то в голову приходит отличная мысль:

```
(definitial list
  (lambda (values) values) )
```

Упражнение 1.7

Естественно, наиболее очевидный способ определить `call/cc` — это использовать `call/cc`. Хитрость здесь в том, чтобы вложенная `call/cc` правильно вызвала передаваемую ей функцию определяемого Лиспа. Для этого её надо вручную заворачивать в `invoke`:

```
(defprimitive call/cc
  (lambda (f)
    (call/cc (lambda (g)
               (invoke f
                (list (lambda (values)
                        (if (= (length values) 1)
                            (g (car values))
                            (wrong "Incorrect arity" g) ) )) ) )) )
  1 )
```

Упражнение 1.8

Здесь возникает та же проблема, что и с `call/cc`: приходится работать одновременно в двух мирах с двумя различными языками. Кроме того, функция `apply` имеет переменную арность, так что вдобавок требуется правильно собрать в список аргументы вызываемой функции (и учесть, что у неё тоже может быть точечный аргумент).

```
(definitial apply
  (lambda (values)
    (if (>= (length values) 2)
        (let ((f (car values))
              (args (let flat ((args (cdr values)))
                      (if (null? (cdr args))
                          (car args)
                          (cons (car args) (flat (cdr args))) ) )) )
          (invoke f args) )
        (wrong "Incorrect arity" 'apply) ) ) )
```

Упражнение 1.9

Просто захватите продолжение вызова интерпретатора и свяжите его с `end`. (К счастью, в Scheme синтаксис активации продолжений не отличается от вызова функций.)

```
(define (chapter1-scheme)
  (define (toplevel)
    (display (evaluate (read) env.global))
    (toplevel) )
  (call/cc (lambda (end)
             (defprimitive end end 1)
             (toplevel) )) )
```

Упражнение 1.10

Конечно, результаты подобных сравнений одновременно зависят как от используемой реализации, так и от рассматриваемых программ. Но в среднем можно сказать, что разница в быстродействии будет порядка 5–15 раз [ITW86].

Даже так, главной задачей этого упражнения было подтолкнуть вас к осознанию того, что `evaluate` написана на фундаментальном Лиспе, поэтому с равным успехом может быть исполнена как интерпретатором Scheme, так и интерпретатором того языка, который сама же определяет, и которым сама же является.

Упражнение 1.11

Так как *список* упорядоченных выражений всегда можно представить в виде пар упорядоченных выражений, то достаточно будет показать, как последовательно вычислить два выражения. Идея состоит хитроумном использовании замыканий для разрешения возможных конфликтов имён.

`(begin` *выражение*$_1$ *выражение*$_2$`)` $\equiv$ `((lambda (void other) (other))`
выражение$_1$
`(lambda ()` *выражение*$_2$`) )`

Упражнение 2.1

На Scheme оно переводится непосредственно как `(cons 1 2)`. Если быть дотошным, то можно сделать так:

```
(define (funcall f . args) (apply f args))
(define (function f) f)
```

Или то же самое с помощью макросов:

```
(define-syntax funcall
  (syntax-rules ()
    ((funcall f arg ...) (f arg ...)) ) )

(define-syntax function
  (syntax-rules ()
    ((function f) f) ) )
```

Упражнение 2.2

Перед ответом на этот вопрос сначала попробуйте ответить на два других:

1) Можно ли ссылаться на функцию `bar` до того, как она была определена?

2) Если `bar` всё же была определена ранее, то как поведёт себя `defun`: выдаст ошибку или переопределит функцию?

А собственно результат исполнения программы зависит от того, что возвращает специальная форма `function`: *саму* функцию `bar` или некоторое значение, связанное с именем `bar` в пространстве функций.

Упражнение 2.3

За вызовы функций отвечает `invoke`, так что достаточно просто научить её не пугаться при виде чисел и списков. Вот так:

```
(define (invoke fn args)
  (cond ((procedure? fn) (fn args))
        ((number? fn)
```

```
        (if (= (length args) 1)
            (if (>= fn 0)
                (list-ref (car args) fn)
                (list-tail (car args) (- fn)) )
            (wrong "Incorrect arity" fn) ) )
       ((pair? fn)
        (map (lambda (f) (invoke f args)) fn) )
       (else (wrong "Cannot apply" fn)) ) )
```

Упражнение 2.4

Сложность здесь в том, что компаратор берётся из определяемого Лиспа и возвращает логические значения оттуда же.

```
(definitial new-assoc/de
  (lambda (values current.denv)
    (if (= 3 (length values))
        (let ((tag        (car values))
              (default    (cadr values))
              (comparator (caddr values)) )
          (let look ((denv current.denv))
            (if (pair? denv)
                (if (eq? the-false-value
                         (invoke comparator (list tag (caar denv))
                                            current.denv ) )
                    (look (cdr denv))
                    (cdar denv) )
                (invoke default (list tag) current.denv) ) ) )
        (wrong "Incorrect arity" 'assoc/de) ) ) )
```

Упражнение 2.5

Функция-обработчик `specific-error` должна будет вывести соответствующее сообщение о неизвестной динамической переменной.

```
(define-syntax dynamic-let
  (syntax-rules ()
    ((dynamic-let () . body)
     (begin . body) )
    ((dynamic-let ((variable value) others ...) . body)
     (bind/de 'variable (list value)
              (lambda () (dynamic-let (others ...) . body)) ) ) ) )

(define-syntax dynamic
  (syntax-rules ()
    ((dynamic variable)
     (car (assoc/de 'variable specific-error)) ) ) )
```

```
(define-syntax dynamic-set!
  (syntax-rules ()
    ((dynamic-set! variable value)
     (set-car! (assoc/de 'variable specific-error) value) ) ) )
```

Упражнение 2.6

Переменная `properties`, замыкаемая обеими функциями, содержит список свойств всех символов.

```
(let ((properties '()))
  (set! putprop
        (lambda (symbol key value)
          (let ((plist (assq symbol properties)))
            (if (pair? plist)
                (let ((couple (assq key (cdr plist))))
                  (if (pair? couple)
                      (set-cdr! couple value)
                      (set-cdr! plist (cons (cons key value)
                                            (cdr plist) )) ) )
                (let ((plist (list symbol (cons key value))))
                  (set! properties (cons plist properties)) ) ) )
          value ) )
  (set! getprop
        (lambda (symbol key)
          (let ((plist (assq symbol properties)))
            (if (pair? plist)
                (let ((couple (assq key (cdr plist))))
                  (if (pair? couple)
                      (cdr couple)
                      #f ) )
                #f ) ) ) ) )
```

Упражнение 2.7

Просто добавьте следующие строки в `evaluate`:

```
...
((label)  ; Синтаксис: (label имя (lambda (аргументы) тело))
 (let* ((name    (cadr e))
        (new-env (extend env (list name) (list 'void)))
        (def     (caddr e))
        (fun     (make-function (cadr def) (cddr def) new-env)) )
   (update! name new-env fun)
   fun ) )
...
```

Упражнение 2.8

Достаточно добавить следующий фрагмент в `f.evaluate`. Обратите внимание на его схожесть с определением `flet`; разница только в окружении, где создаются локальные функции.

```
...
((labels)
 (let ((new-fenv (extend fenv
                         (map car (cadr e))
                         (map (lambda (def) 'void) (cadr e)) )))
   (for-each (lambda (def)
               (update! (car def)
                        new-fenv
                        (f.make-function (cadr def) (cddr def)
                                         env new-fenv ) ) )
             (cadr e) )
   (f.eprogn (cddr e) env new-fenv) ) )
...
```

Упражнение 2.9

Так как форма `let` сохраняет неопределённый порядок вычислений, то её следует использовать для вычисления значений переменных формы `letrec`. Связывание же этих переменных с полученными значениями необходимо выполнять отдельно. Имена для всех этих $temp_i$ можно получить или с помощью механизма макрогигены, или просто кучей вызовов `gensym`.

```
(let ((переменная_1 'void)
      ...
      (переменная_n 'void) )
  (let ((temp_1 выражение_1)
        ...
        (temp_n выражение_n) )
    (set! переменная_1 temp_1)
    ...
    (set! переменная_n temp_n)
    тело ) )
```

Упражнение 2.10

Вот вам вариант для бинарных функций. η-конверсия была модифицирована соответствующим образом.

```
(define fix2
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda (x y) (((w w) f) x y))) ) )))
    (d d) ) )
```

После этого довольно легко догадаться, как сделать n-арную версию:

```
(define fixN
  (let ((d (lambda (w)
             (lambda (f)
               (f (lambda args (apply ((w w) f) args))) ) )))
    (d d) ) )
```

Упражнение 2.11

Ещё одно умственное усилие — и вы увидите, что предыдущее определение `fixN` легко расширяется:

```
(define 2fixN
  (let ((d (lambda (w)
             (lambda (f*)
               (list ((car f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) )
                     ((cadr f*)
                      (lambda a (apply (car ((w w) f*)) a))
                      (lambda a (apply (cadr ((w w) f*)) a)) ) ) ) )))
    (d d) ) )
```

После этого остаётся понять, когда именно должен быть вычислен терм `((w w) f)`, и можно будет написать правильную универсальную версию:

```
(define NfixN
  (let ((d (lambda (w)
             (lambda (f*)
               (map (lambda (f)
                      (apply f (map (lambda (i)
                                      (lambda a
                                        (apply (list-ref ((w w) f*) i)
                                               a ) ) )
                                    (iota 0 (length f*)) )) )
                    f* ) ) )))
    (d d) ) )
```

Внимание: порядок функций важен. Если определение `odd?` идёт первым в списке функционалов, то именно эта функция будет связана с их первыми аргументами.

Функция `iota` аналогична одноимённому примитиву ι языка APL:

```
(define (iota start end)
  (if (< start end)
      (cons start (iota (+ 1 start) end))
      '() ) )
```

Упражнение 2.12

[Bar84] приписывает эту функцию Яну Виллему Клопу. Можете проверить, что `((klop meta-fact) 5)` действительно возвращает `120`.

Так как все внутренние переменные `s`, `c`, `h`, `e`, `m` связываются с одной `r`, то их порядок в аппликации `(m e c h e s)` не имеет значения. Важно только их количество. Вернее, согласованная арность: можно оставить одну переменную `w`, а можно использовать хоть весь алфавит — в любом случае получится Y.

Упражнение 2.13

Абсолютно неожиданный ответ: 120. Вам ведь понравилось выражать рекурсию с помощью самоприменения, правда? Данное определение можно записать немного по-другому, используя вложенные `define`:

```
(define (factfact n)
  (define (internal-fact f n)
    (if (= n 0) 1
        (* n (f f (- n 1))) ) )
  (internal-fact internal-fact n) )
```

Упражнение 3.1

Эту форму можно было назвать `(the-current-continuation)`, так как она возвращает собственное продолжение. Давайте разберёмся, как у неё это получается. Для понятности будем нумеровать используемые продолжения и функции, а `call/cc` сократим до просто `cc`. Итак, вычисляемое выражение: ${}_{k_0}(\mathtt{cc}_1\ \ \mathtt{cc}_2)$. k_0 — это продолжение данных вычислений. Определение `call/cc`:

${}_{k}(\mathtt{call/cc}\ \varphi) \quad \to \quad {}_{k}(\varphi\ k)$

Следовательно, ${}_{k_0}(\mathtt{cc}_1\ \ \mathtt{cc}_2)$ становится ${}_{k_0}(\mathtt{cc}_2\ \ k_0)$, которое в свою очередь переходит в ${}_{k_0}(k_0\ \ k_0)$, которое, очевидно, возвращает k_0.

Упражнение 3.2

Используя нотацию предыдущего упражнения, запишем: ${}_{k_0}((\mathtt{cc}_1\ \ \mathtt{cc}_2)\ (\mathtt{cc}_3\ \mathtt{cc}_4))$. Для простоты будем считать, что термы аппликаций вычисляются слева направо. Тогда исходное выражение эквивалентно ${}_{k_0}({}_{k_1}(\mathtt{cc}_1\ \ \mathtt{cc}_2)\ (\mathtt{cc}_3\ \mathtt{cc}_4))$, где k_1 равно $\lambda\varphi.{}_{k_0}(\varphi\ \ {}_{k_2}(\mathtt{cc}_3\ \ \mathtt{cc}_4))$, а k_2 это $\lambda\varepsilon.{}_{k_0}(k_1\ \ \varepsilon)$. Вычисление первого терма приводит к ${}_{k_0}(k_1\ \ k_2)$, а вычисление этого — к ${}_{k_0}(k_2\ \ {}_{k_2'}(\mathtt{cc}_3\ \ \mathtt{cc}_4))$, где k_2' равно $\lambda\varepsilon.{}_{k_0}(k_2\ \ \varepsilon)$. Эта форма вычисляется в ${}_{k_0}(k_1\ \ k_2')$, что впоследствии приводит к ${}_{k_0}(k_1\ \ k_2'')$, и так далее. Как видите, вычисления зацикливаются. Можно доказать, что результат не зависит от порядка вычисления термов аппликаций. Вполне вероятно, что это самая короткая программа на Лиспе, выражающая бесконечный цикл.

Упражнение 3.3

Метки разделяют тело `tagbody` на отдельные последовательности выражений. Эти последовательности оборачиваются в функции и помещаются в гигантскую форму `labels`. Формы `go` преобразуются в вызовы соответствующих функций, но данные вызовы выполняются специальным образом, чтобы `go` получила правильное продолжение. В итоге `tagbody` становится этим:

```
(block EXIT
  (let (LABEL (TAG (list 'tagbody)))
    (labels ((INIT  () выражения0... (метка1))
             (метка1 () выражения1... (метка2))
             ...
             (меткаn () выраженияn... (return-from EXIT nil)) )
      (setq LABEL (function INIT))
      (while #t
        (setq LABEL (catch TAG (funcall LABEL))) ) ) ) )
```

Формы `(go` *метка*`)` становятся `(throw TAG` *метка*`)`, а `(return` *значение*`)` превращается в `(return-from EXIT` *значение*`)`. Имена переменных, записанные `ПРОПИСНЫМИ` буквами, не должны конфликтовать с переменными, используемыми в теле `tagbody`.

Такое сложное представление `go` необходимо для того, чтобы обеспечить переходам правильное продолжение: в форме `(bar (go L))` не надо вызывать функцию `bar` после того, как `(go L)` вернёт значение. Если этого не сделать, то вот такая программа будет вести себя неправильно:

```
(tagbody  A (return (+ 1 (catch 'foo (go B))))
          B (* 2 (throw 'foo 5)) )
```

См. также [Bak92c].

Упражнение 3.4

Введите новый класс функций:

```
(define-class function-with-arity function (arity))
```

Затем измените обработку `lambda`-форм, чтобы они возвращали именно такие объекты:

```
(define (evaluate-lambda n* e* r k)
  (resume k (make-function-with-arity n* e* r (length n*))) )
```

И, наконец, реализуйте оптимизированный протокол вызова данных функций:

```
(define-method (invoke (f function-with-arity) v* r k)
  (if (= (function-with-arity-arity f) (length v*))
      (let ((env (extend-env (function-env f)
                             (function-variables f) v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Упражнение 3.5

```
(definitial apply
  (make-primitive 'apply
   (lambda (v* r k)
     (if (>= (length v*) 2)
         (let ((f (car v*))
               (args (let flat ((args (cdr v*)))
                       (if (null? (cdr args))
                           (car args)
                           (cons (car args) (flat (cdr args))) ) )) )
           (invoke f args r k) )
         (wrong "Incorrect arity" 'apply) ) ) ) )
```

Упражнение 3.6

Определите новый класс функций по аналогии с упражнением 3.4.

```
(define-class function-nary function (arity))

(define (evaluate-lambda n* e* r k)
  (resume k (make-function-nary n* e* (length n*))) )

(define-method (invoke (f function-nary) v* r k)
  (define (extend-env env names values)
    (if (pair? names)
        (make-variable-env
         (extend-env env (cdr names) (cdr values))
         (car names)
         (car values) )
        (make-variable-env env names values) ) )
  (if (>= (length v*) (function-nary-arity f))
      (let ((env (extend-env (function-env f)
                             (function-variables f)
                             v* )))
        (evaluate-begin (function-body f) env k) )
      (wrong "Incorrect arity" (function-variables f) v*) ) )
```

Упражнение 3.7

Реализуйте циклическое выполнение `evaluate` с помощью начального продолжения:

```
(define (chapter3-interpreter-2)
  (letrec ((k.init (make-bottom-cont
                    'void (lambda (v) (display v)
                                      (toplevel) ) ))
           (toplevel (lambda () (evaluate (read r.init k.init)))) )
    (toplevel) ) )
```

Упражнение 3.8

Определите соответствующий класс значений-продолжений. Он должен инкапсулировать продолжения языка реализации и предоставлять метод для их активации. `call/cc` теперь будет возвращать именно такие объекты.

```
(define-class reified-continuation value (k))

(definitial call/cc
  (make-primitive 'call/cc
   (lambda (v* r k)
     (if (= 1 (length v*))
         (invoke (car v*) (list (make-reified-continuation k)) r k)
         (wrong "Incorrect arity" 'call/cc v*) ) ) ) )

(define-method (invoke (f reified-continuation) v* r k)
  (if (= 1 (length v*))
      (resume (reified-continuation-k f) (car v*))
      (wrong "Continuations expect one argument" v* r k) ) )
```

Упражнение 3.9

Вычисление функции заканчивается возвратом значения. Перехватывайте все попытки вернуть его.

```
(defun eternal-return (thunk)                                  Common Lisp
  (labels ((loop ()
             (unwind-protect (thunk)
               (loop) ) ))
    (loop) ) )
```

Упражнение 3.10

Значения этих выражений: 33 и 44 соответственно. Функция `make-box` создаёт *коробку*, которая может хранить в себе одно значение. Причём это значение можно изменять без видимых побочных эффектов. Достигается такое поведение с помощью `call/cc` и `letrec`. Если вспомнить, что `letrec` эквивалентна комбинации `let` и `set!`, то станет понятнее, каким образом нам удаётся получить такой эффект. Полноценные продолжения Scheme, способные сколько угодно раз возвращаться к прерванным вычислениям, позволяют отделить неявную `set!`-часть формы `letrec` от её `let`-части.

Упражнение 3.11

Сначала сделайте `evaluate` обобщённой:

```
(define-generic (evaluate (e) r k)
  (wrong "Not a program" e) )
```

Затем напишите для неё методы, вызывающие соответствующие функции:

```
(define-method (evaluate (e quotation) r k)
  (evaluate-quote (quotation-value e) r k) )

(define-method (evaluate (e assignment) r k)
  (evaluate-set! (assignment-name e)
                 (assignment-form e)
                 r k ) )
...
```

Также вам понадобятся новые классы объектов для представления различных частей программ:

```
(define-class program    Object  ())
(define-class quotation  program (value))
(define-class assignment program (name form))
...
```

Всё, теперь остаётся только определить функцию, преобразующую текст программ в объекты класса `program`. Эта функция, называемая `objectify`, рассматривается в разделе 9.11.1.

Упражнение 3.12

Функция `throw` определяется вот так:

```
(definitial throw
  (make-primitive 'throw
   (lambda (v* r k)
     (if (= 2 (length v*))
         (catch-lookup k (car v*)
                       (make-throw-cont k
                        `(quote ,(cadr v*)) r ) )
         (wrong "Incorrect arity" 'throw v*) ) ) ) )
```

Вместо того, чтобы определять новый метод для `catch-lookup`, мы просто подсунули ей фальшивое продолжение, чтобы заставить интерпретатор вести себя ожидаемым образом: вычислить и вернуть второй аргумент `throw`, когда найдётся соответствующая форма `catch`.

Упражнение 3.13

CPS-код медленнее обычного, так как он вынужден постоянно создавать замыкания для явного представления продолжений.

Между прочим, CPS-преобразование не идемпотентно; то есть, применив его к программе, уже переписанной в стиле передачи продолжений, мы получим ещё одну, третью версию той же программы. Рассмотрим, например, определение факториала:

```
(define (cps-fact n k)
  (if (= n 0) (k 1)
      (cps-fact (- n 1) (lambda (v) (k (* n v)))) ) )
```

Очевидно, что `k` — это просто аргумент функции `cps-fact`. Он может быть вообще чем угодно. В том числе и таким продолжением:

```
(call/cc (lambda (k) (* 2 (cps-fact 4 k)))) → 24
```

Упражнение 3.14

Функцию `the-current-continuation` также можно определить подобно упражнению 3.1.

```
(define (cc f)
  (let ((reified? #f))
    (let ((k (the-current-continuation)))
      (if reified? k
          (begin (set! reified? #t)
                 (f k) ) ) ) ) )
```

Большое спасибо Люку Моро за эту пару упражнений [Mor94].

Упражнение 4.1

Количество способов написания этой функции огромно. Например, можно возвращать промежуточные результаты или использовать продолжения:

```
(define (min-max1 tree)
  (define (mm tree)
    (if (pair? tree)
        (let ((a (mm (car tree)))
              (b (mm (cdr tree))) )
          (list (min (car a) (car d))
                (max (cadr a) (cadr d)) ) )
        (list tree tree) ) )
  (mm tree) )

(define (min-max2 tree)
  (define (mm tree k)
    (if (pair? tree)
        (mm (car tree)
            (lambda (mina maxa)
              (mm (cdr tree)
                  (lambda (mind maxd)
                    (k (min mina mind)
                       (max maxa maxd) ) ) ) ) )
        (k tree tree) ) )
  (mm tree list) )
```

Первый вариант в процессе работы постоянно создаёт и тут же уничтожает кучу списков. Ситуацию можно поправить с помощью известной оптимизации, называемой *deforestation* [Wad88]. Она позволяет избавиться от лишних промежуточных структур данных. Второй вариант в этом плане ничем не лучше: просто вместо списков здесь замыкания. Исходная версия гораздо быстрее любого из них (но она использует «невыносимо отвратительные» побочные эффекты).

Упражнение 4.2

Функции начинаются на `q`, чтобы избежать путаницы.

```
(define (qons a d) (lambda (msg) (msg a d)))
(define (qar pair) (pair (lambda (a d) a)))
(define (qdr pair) (pair (lambda (a d) d)))
```

Упражнение 4.3

Идея в том, что две точечные пары идентичны, если модификация одной из них приводит к изменениям в другой.

```
(define (pair-eq? a b)
  (let ((tag (list 'tag))
        (old-car (car a)) )
    (set-car! a tag)
    (let ((result (eq? (car b) tag)))
      (set-car! a old-car)
      result ) ) )
```

Упражнение 4.4

Добавляете анализ новой специальной формы в `evaluate`:

```
...
((or) (evaluate-or (cadr e) (caddr e) r s k))
...
```

После этого определяете её как-то так:

```
(define (evaluate-or e1 e2 r s k)
  (evaluate e1 r s (lambda (v ss)
                     (((v 'boolify)
                       (lambda () (k v ss))
                       (lambda () (evaluate e2 r k s)) )) )) )
```

Суть в том, что вычисление альтернативной ветки β производится в старой памяти `s`, а не в новой `ss`.

Упражнение 4.5

Вообще-то такая формулировка задания допускает разночтения: можно ведь возвращать то значение переменной, которое она имела до вычисления её нового значения, а можно вернуть и то, каким оно стало после.

```
(define (pre-evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k (ss (r n)) (update ss (r n) v)) ) ) )

(define (post-evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k (s (r n)) (update ss (r n) v)) ) ) )
```

Это важно. Например, значение данного выражения зависит от реализации:

```
(let ((x 1))
  (set! x (set! x 2)) )
```

Упражнение 4.6

Основная сложность в `apply` — это правильно обработать список её аргументов, созданный интерпретатором определяемого языка.

```
(definitial apply
  (create-function
   -11 (lambda (v* s k)
         (define (first-pairs v*)
           ;; (assume (pair? v*))
           (if (pair? (cdr v*))
               (cons (car v*) (first-pairs (cdr v*)))
               '() ) )
         (define (terms-of v s)
           (if (eq? (v 'type) 'pair)
               (cons (s (v 'car)) (terms-of (s (v 'cdr)) s))
               '() ) )
         (if (>= (length v*) 2)
             (if (eq? ((car v*) 'type) 'function)
                 (((car v*) 'behavior)
                  (append (first-pairs (cdr v*))
                          (terms-of (car (last-pair (cdr v*))) s) )
                  s k )
                 (wrong "First argument not a function") )
             (wrong "Incorrect arity") ) ) ) )
```

Функция `call/cc` сохраняет каждое продолжение в собственной ячейке памяти, чтобы сделать их уникальными.

```
(definitial call/cc
  (create-function
   -13 (lambda (v* s k)
         (if (= 1 (length v*))
             (if (eq? ((car v*) 'type) 'function)
                 (allocate 1 s
                  (lambda (a* ss)
                    (((car v*) 'behavior)
                     (list (create-function
                            (car a*)
                            (lambda (vv* sss kk)
                              (if (= 1 (length vv*))
                                  (k (car vv*) sss)
                                  (wrong "Incorrect arity") ) ) ))
                           ss k ) ) )
                 (wrong "Argument not a function") )
             (wrong "Incorrect arity") ) ) ) )
```

Упражнение 4.7

Сложность здесь состоит в проверке совместимости количества фактически полученных аргументов с арностью вызываемой функции, а также в преобразовании списков и значений при передаче их между языками.

```
(define (evaluate-nlambda n* e* r s k)
  (define (arity n*)
    (cond ((pair? n*) (+ 1 (arity (cdr n*))))
          ((null? n*) 0)
          (else       1) ) )

  (define (update-environment r n* a*)
    (cond ((pair? n*) (update-environment
                        (update r (car n*) (car a*))
                        (cdr n*) (cdr* a) ))
          ((null? n*) r)
          (else (update r n* (car a*))) ) )

  (define (update-store s a* v* n*)
    (cond ((pair? n*) (update-store (update s (car a*) (car v*))
                                    (cdr a*) (cdr v*) (cdr n*) ))
          ((null? n*) s)
          (else (allocate-list v* s (lambda (v ss)
                                      (update ss (car a*) v) ))) ) )
  (allocate 1 s
    (lambda (a* ss)
      (k (create-function
          (car a*)
```

```
            (lambda (v* s k)
              (if (compatible-arity? n* v*)
                  (allocate (arity n*) s
                   (lambda (a* ss)
                     (evaluate-begin e*
                                     (update-environment r n* a*)
                                     (update-store ss a* v n*)
                                     k ) ) )
                  (wrong "Incorrect arity") ) ) )
          ss ) ) ) )

(define (compatible-arity? n* v*)
  (cond ((pair? n*) (and (pair? v*)
                         (compatible-arity? (cdr n*) (cdr v*)) ))
        ((null? n*) (null? v*))
        ((symbol? n*) #t) ) )
```

Упражнение 5.1

Доказывается индукцией по количеству термов аппликации.

Упражнение 5.2

$\mathcal{L}[\![\texttt{(label } \nu \ \pi\texttt{)}]\!]\rho = (\mathsf{Y}\ \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \to \varepsilon]))$

Упражнение 5.3

$\mathcal{E}[\![\texttt{(dynamic } \nu\texttt{)}]\!]\rho\delta\kappa\sigma =$
let $\varepsilon = (\delta\ \nu)$
in if $\varepsilon =$ *no-dynamic-binding*
  then let $\alpha = (\gamma\ \nu)$
    in if $\alpha =$ *no-global-binding*
      then *wrong* `"No such variable"`
      else $(\kappa\ (\sigma\ \alpha)\ \sigma)$
      endif
  else $(\kappa\ \varepsilon\ \sigma)$
  endif

Упражнение 5.4

Этот макрос помещает вычисление каждого терма в собственное замыкание, после чего выполняет все эти вычисления в произвольном порядке, определяемом функцией `determine!`.

```
(define-syntax unordered
  (syntax-rules ()
    ((unordered f) (f))
```

```
    ((unordered f arg ...)
     (determine! (lambda () f) (lambda () arg) ...) ) ) )

(define (determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (random-permutation (length thunks))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (cdr permut)) )
          (apply (car results) (cdr results)) ) ) ) )
```

Заметьте, что порядок выбирается перед началом вычислений, так что такое определение не совсем идентично денотации, приведённой в этой главе. Если функция `random-permutation` определена вот так:

```
(define (random-permutation n)
  (shuffle (iota 0 n)) )
```

то последовательность вычислений выбирается действительно динамически:

```
(define (d.determine! . thunks)
  (let ((results (iota 0 (length thunks))))
    (let loop ((permut (random-permutation (length thunks))))
      (if (pair? permut)
          (begin (set-car! (list-tail results (car permut))
                           (force (list-ref thunks (car permut))) )
                 (loop (shuffle (cdr permut))) )
          (apply (car results) (cdr results)) ) ) ) )
```

Упражнение 6.1

Самый простой способ — это добавить `CHECKED-GLOBAL-REF` ещё один аргумент с именем соответствующей переменной:

```
(define (CHECKED-GLOBAL-REF- i n)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable" n)
          v ) ) ) )
```

Однако такой подход нерационально расходует память и дублирует информацию. Более правильным решением будет создать специальную таблицу символов для хранения соответствий между адресами переменных и их именами.

```
(define sg.current.names (list 'foo))
(define (standalone-producer e)
  (set! g.current (original.g.current))
```

```
    (let* ((m (meaning e r.init #t))
           (size (length g.current))
           (global-names (map car (reverse g.current))) )
      (lambda ()
        (set! sg.current (make-vector size undefined-value))
        (set! sg.current.names global-names)
        (set! *env* sr.init)
        (m) ) ) )

(define (CHECKED-GLOBAL-REF+ i)
  (lambda ()
    (let ((v (global-fetch i)))
      (if (eq? v undefined-value)
          (wrong "Uninitialized variable"
                 (list-ref sg.current.names i) )
          v ) ) ) )
```

Упражнение 6.2

Функция `list` — это, конечно же, просто `(lambda l l)`. Вам надо только выразить это определение с помощью комбинаторов:

```
(definitial list ((NARY-CLOSURE (SHALLOW-ARGUMENT-REF 0) 0)))
```

Упражнение 6.3

Всё просто: достаточно переопределить каждый комбинатор k как `(lambda args '(`k `. ,args))` и распечатать результат предобработки.

Упражнение 6.4

Решение в лоб: вычислять термы аппликации справа налево:

```
(define (FROM-RIGHT-STORE-ARGUMENT m m* index)
  (lambda ()
    (let* ((v* (m*))
           (v  (m)) )
      (set-activation-frame-argument! v* index v)
      v* ) ) )

(define (FROM-RIGHT-CONS-ARGUMENT m m* arity)
  (lambda ()
    (let* ((v* (m*))
           (v  (m)) )
      (set-activation-frame-argument!
       v* arity (cons v (activation-frame-argument v* arity)) )
      v* ) ) )
```

Также можно изменить не порядок вычисления аргументов, а определение `meaning*`, чтобы она создавала запись активации первой. В любом случае эффективнее будет сначала вычислить функциональный терм (порядок вычисления остальных аргументов здесь не важен), так как это позволяет узнать истинную арность вызываемого замыкания и сразу создавать запись активации правильного размера.

Упражнение 6.5

Определите синтаксис новой специальной формы в `meaning`:

```
... ((redefine) (meaning-redefine (cadr e))) ...
```

Затем реализуйте её предобработку:

```
(define (meaning-redefine n)
  (let ((kind1 (global-variable? g.init n)))
    (if kind1
        (let ((value (vector-ref sg.init (cdr kind)))
              (kind2 (global-variable? g.current n)) )
          (if kind2
              (static-wrong "Already redefined variable" n)
              (let ((index (g.current-extend! n)))
                (vector-set! sg.current index value) ) ) )
        (static-wrong "Can't redefine variable" n) )
    (lambda () 2001) ) )
```

Подобные переопределения производятся во время предобработки, ещё до исполнения программы. Возвращаемое значение формы `redefine` не важно.

Упражнение 6.6

Вызов функции без аргументов не требует выделения памяти под переменные, то есть расширения текущего окружения. Каждый дополнительный уровень окружения увеличивает стоимость обращений к свободным переменным замыканий, что сказывается на быстродействии. Реализуйте новый комбинатор и добавьте в определение `meaning-fix-abstraction` обработку соответствующего специального случая.

```
(define (THUNK-CLOSURE m+)
  (let ((arity+1 (+ 0 1)))
    (lambda ()
      (define (the-function v* sr)
        (if (= (activation-frame-argument-length v*) arity+1)
            (begin (set! *env* sr)
                   (m+) )
            (wrong "Incorrect arity") ) )
      (make-closure the-function *env*) ) ) )
```

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let ((arity (length n*)))
    (if (= arity 0)
        (let ((m+ (meaning-sequence e+ r #t)))
          (THUNK-CLOSURE m+) )
        (let* ((r2 (r-extend* r n*))
               (m+ (meaning-sequence e+ r2 #t)) )
          (FIX-CLOSURE m+ arity) ) ) ) )
```

Упражнение 7.1

Сначала создайте новый регистр:

```
(define *dynenv* -1)
```

Затем сохраняйте его вместе с остальным окружением:

```
(define (preserve-environment)
  (stack-push *dynenv*)
  (stack-push *env*) )

(define (restore-environment)
  (set! *env* (stack-pop))
  (set! *dynenv* (stack-pop)) )
```

Теперь динамическое окружение извлекается элементарно; лишь несколько изменилась работа со стеком:

```
(define (search-dynenv-index)
  *dynenv* )

(define (pop-dynamic-binding)
  (stack-pop)
  (stack-pop)
  (set! *dynenv* (stack-pop)) )

(define (push-dynamic-binding index value)
  (stack-push *dynenv*)
  (stack-push value)
  (stack-push index)
  (set! *dynenv* (- *stack-index* 1)) )
```

Упражнение 7.2

Сама функция-то простая:

```
(definitial load
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (make-primitive
```

```
(lambda ()
  (if (= arity+1 (activation-frame-argument-length *val*))
      (let ((filename (activation-frame-argument *val* 0)))
        (set! *pc* (install-object-file! filename)) )
      (signal-exception
       #t (list "Incorrect arity" 'load) ) ) ) ) ) )
```

Но вот при её использовании возникают определённые сложности. Всё дело в продолжениях. Допустим, с помощью `load` загружается следующий файл:

```
(display 'attention)
(call/cc (lambda (k) (set! *k* k)))
(display 'caution)
```

Что случится, если после этого активировать продолжение `*k*`? Правильно, выведется символ `caution`! А потом?

Кроме того, определения глобальных переменных из загружаемого файла не переходят в текущий (что, согласитесь, будет сюрпризом для функций, которые от них зависят).

Упражнение 7.3

Всё просто:

```
(definitial global-value
  (let* ((arity 1)
         (arity+1 (+ 1 arity)) )
    (define (get-index name)
      (let ((where (memq name sg.current.names)))
        (if where
            (- (length where) 1)
            (signal-exception
             #f (list "Undefined global variable" name) ) ) ) )
    (make-primitive
     (lambda ()
       (if (= arity+1 (activation-frame-argument-length *val*))
           (let* ((name (activation-frame-argument *val* 0))
                  (i (get-index name)) )
             (set! *val* (global-fetch i))
             (when (eq? *val* undefined-value)
               (signal-exception #f (list "Uninitialized variable" i)) )
             (set! *pc* (stack-pop)) )
           (signal-exception
            #t (list "Incorrect arity" 'global-value) ) ) ) ) ) )
```

Во время вызова этой функции переменная может как просто не существовать, так и ещё не иметь значения. Оба этих случая необходимо проверять.

Упражнение 7.4

Для начала добавьте в `run-machine` инициализацию вектора текущего состояния динамического окружения:

```
... (set! *dynamics* (make-vector (+ 1 (length dynamics))
                                  undefined-value )) ...
```

После чего переопределите функции-аксессоры на новый лад:

```
(define (find-dynamic-value index)
  (let ((v (vector-ref *dynamics* index)))
    (if (eq? v undefined-value)
        (signal-exception #f (list "No such dynamic binding" index))
        v ) ) )

(define (push-dynamic-binding index value)
  (stack-push (vector-ref *dynamics* index))
  (stack-push index)
  (vector-set! *dynamics* index value) )

(define (pop-dynamic-binding)
  (let* ((index (stack-pop))
         (old-value (stack-pop)) )
    (vector-set! *dynamics* index old-value) ) )
```

Увы, но такое решение в общем случае неверно. В стеке сейчас сохраняются только предыдущие значения динамических переменных, но не текущие. Следовательно, любой переход или активация продолжения приведут к неправильному состоянию динамического окружения, так как мы не сможем восстановить значение `*dynamics*` на момент входа в форму `bind-exit` или `call/cc`. Чтобы реализовать данное поведение, необходима форма `unwind-protect`; ну, или можно отказаться от такого подхода в пользу дальнего связывания, где подобные проблемы не возникают в принципе.

Упражнение 7.5

С помощью следующей функции можно выразить даже взаимные переименования вида `((fact fib) (fib fact))`. Но не стоит этим злоупотреблять.

```
(define (build-application-with-renaming-variables
         new-application-name application-name substitutions )
  (if (probe-file application-name)
      (call-with-input-file application-name
        (lambda (in)
          (let* ((dynamics     (read in))
                 (global-names (read in))
                 (constants    (read in))
                 (code         (read in))
                 (entries      (read in)) )
```

```
            (close-input-port in)
            (write-result-file
             new-application-name
             (list ";;; Renamed variables from " application-name)
             dynamics
             (let sublis ((global-names global-names))
               (if (pair? global-names)
                   (cons (let ((s (assq (car global-names)
                                        substitutions )))
                           (if (pair? s)
                               (cadr s)
                               (car global-names) ) )
                         (sublis (cdr global-names)) )
                   global-names ) )
             constants
             code
             entries ) ) ) )
      (signal-exception #f (list "No such file" application-name)) ) )
```

Упражнение 7.6

Сделать это просто, только не перепутайте коды инструкций и смещения!

```
(define-instruction (CHECKED-GLOBAL-REF i) 8
  (set! *val* (global-fetch i))
  (if (eq? val undefined-value)
      (signal-exception #t (list "Uninitialized variable" i))
      (vector-set! *code* (- *pc* 2) 7) ) )
```

Упражнение 8.1

Она может не волноваться об этом, потому как сравнивает переменные не по именам. Такой подход правильно работает даже для списков с циклами.

Упражнение 8.2

Вот вам подсказка:

```
(define (prepare e)
  (eval/ce `(lambda () ,e)) )
```

Упражнение 8.3

```
(define (eval/at e)
  (let ((g (gensym)))
    (eval/ce `(lambda (,g) (eval/ce ,g))) ) )
```

Упражнение 8.4

Да, определив специальный обработчик исключений:

```
(set! variable-defined?
      (lambda (env name)
        (bind-exit (return)
          (monitor (lambda (c ex) (return #f))
            (eval/b name env)
            #t ) ) ) ) )
```

Упражнение 8.5

Реализацию специальной формы `monitor`, которая используется в рефлексивном интерпретаторе, мы молча пропустим, так как она принципиально непереносима. В конце концов, если не делать ошибок, то `monitor` эквивалентна `begin`. Строго говоря, остальной код, что следует далее, тоже не совсем легален, так как использует переменные с именами специальных форм. Однако, большинство реализаций Scheme допускают такие вольности.

Форма `the-environment`, захватывающая привязки:

```
(define-syntax the-environment
  (syntax-rules ()
    ((the-environment)
     (capture-the-environment make-toplevel make-flambda flambda?
      flambda-behavior prompt-in prompt-out exit it extend error
      global-env toplevel eval evlis eprogn reference quote if set!
      lambda flambda monitor ) ) ) )

(define-syntax capture-the-environment
  (syntax-rules ()
    ((capture-the-environment word ...)
     (lambda (name . value)
       (case name
         ((word) ((handle-location word) value)) ...
         ((display) (if (pair? value)
                        (wrong "Immutable" 'display)
                        show ))
         (else (if (pair? value)
                   (set-top-level-value! name (car value))
                   (top-level-value name) )) ) ) ) ) )

(define-syntax handle-location
  (syntax-rules ()
    ((handle-location name)
     (lambda (value)
       (if (pair? value) (set! name (car value))
           name ) ) ) ) )
```

Функции `variable-defined?`, `variable-value` и `set-variable-value!`, манипулирующие захваченными полноценными окружениями:

```
(define undefined (cons 'un 'defined))

(define-class Envir Object
  ( name value next ) )

(define (enrich env . names)
  (let enrich ((env env) (names names))
    (if (pair? names)
        (enrich (make-Envir (car names) undefined env)
                (cdr names) )
        env ) ) )

(define (variable-defined? name env)
  (if (Envir? env)
      (or (eq? name (Envir-name env))
          (variable-defined? name (Envir-next env)) )
      #f ) )

(define (variable-value name env)
  (if (Envir? env)
      (if (eq? name (Envir-name env))
          (let ((value (Envir-value env)))
            (if (eq? value undefined)
                (error "Uninitialized variable" name)
                value ) )
          (variable-value name (Envir-next env)) )
      (env name) ) )
```

Как видите, окружения — это связные списки, заканчивающиеся замыканием. Теперь рефлексивный интерпретатор может быть запущен!

Упражнение 9.1

Используйте гигиеничные макросы Scheme:

```
(define-syntax repeat1
  (syntax-rules (:while :unless :do)
    ((_ :while p :unless q :do body ...)
     (let loop ()
       (if p (begin (if (not q) (begin body ...))
                    (loop) )) ) ) ) )
```

Как вариант, можно всё сделать вручную с помощью `define-abbreviation`:

```
(with-aliases ((+let let) (+begin begin) (+when when) (+not not))
  (define-abbreviation (repeat2 . params)
    (let ((p    (list-ref  params 1))
```

```
        (q    (list-ref  params 3))
        (body (list-tail params 5))
        (loop (gensym)) )
    `(,+let ,loop ()
        (,+when ,p (,+begin (,+when (,+not ,q) . ,body)
                            (,loop) )) ) ) ) )
```

Упражнение 9.2

Вся хитрость в том, как представить числа с помощью одних только макроопределений. Один из вариантов — это использовать списки такой же длины, что и представляемое ими число. Тогда во время исполнения программы можно будет получить нормальные числа с помощью функции `length`.

```
(define-syntax enumerate
  (syntax-rules ()
    ((enumerate) (display 0))
    ((enumerate e1 e2 ...)
     (begin (display 0)
            (enumerate-aux e1 (e1) e2 ...) ) ) ) )

(define-syntax enumerate-aux
  (syntax-rules ()
    ((enumerate-aux e1 len) (begin (display e1)
                                   (display (length 'len)) ))
    ((enumerate-aux e1 len e2 e3 ...)
     (begin (display e1)
            (display (length 'len))
            (enumerate-aux e2 (e2 . len) e3 ...) ) ) ) )
```

Упражнение 9.3

Достаточно переопределить функцию `make-macro-environment` так, чтобы она использовала текущий уровень, а не создавала следующий:

```
(define (make-macro-environment current-level)
  (let ((metalevel (delay current-level)))
    (list (make-Magic-Keyword 'eval-in-abbreviation-world
           (special-eval-in-abbreviation-world metalevel) )
          (make-Magic-Keyword 'define-abbreviation
           (special-define-abbreviation metalevel) )
          (make-Magic-Keyword 'let-abbreviation
           (special-let-abbreviation metalevel) )
          (make-Magic-Keyword 'with-aliases
           (special-with-aliases metalevel) ) ) ) )
```

Упражнение 9.4

Написать такой конвертер проще пареной репы. Единственный интересный момент — это сборка списка аргументов функции. Здесь используется А-список для хранения соответствий между аргументами и их именами.

```
(define-generic (->Scheme (e) r))

(define-method (->Scheme (e Alternative) r)
  `(if ,(->Scheme (Alternative-condition e) r)
       ,(->Scheme (Alternative-consequent e) r)
       ,(->Scheme (Alternative-alternant e) r) ) )

(define-method (->Scheme (e Local-Assignment) r)
  `(set! ,(->Scheme (Local-Assignment-reference e) r)
         ,(->Scheme (Local-Assignment-form e) r) ) )

(define-method (->Scheme (e Reference) r)
  (variable->Scheme (Reference-variable e) r) )

(define-method (->Scheme (e Function) r)
  (define (renamings-extend r variables names)
    (if (pair? names)
        (renamings-extend (cons (cons (car variables) (car names)) r)
                          (cdr variables) (cdr names) )
        r ) )
  (define (pack variables names)
    (if (pair? variables)
        (if (Local-Variable-dotted? (car variables))
            (car names)
            (cons (car names) (pack (cdr variables) (cdr names))) )
        '() ) )
  (let* ((variables (Function-variables e))
         (new-names (map (lambda (v) (gensym))
                         variables ))
         (newr (renamings-extend r variables new-names)) )
    `(lambda ,(pack variables new-names)
       ,(->Scheme (Function-body e) newr) ) ) )

(define-generic (variable->Scheme (e) r))
```

Упражнение 9.5

В текущем состоянии MEROONET действительно существует в двух мирах одновременно. Например, функция `register-class` вызывается как во время раскрытия макросов, так и в процессе динамической загрузки файлов.

Упражнение 10.1

Во-первых, доработайте функцию `SCM_invoke`: возьмите за основу протокол вызова примитивов и сделайте подобную специализацию для замыканий. Во-вторых, не забудьте передать замыкание самому себе в качестве первого аргумента. В-третьих, специализируйте также кодогенераторы для замыканий, чтобы сигнатуры соответствующих функций совпадали с тем, чего ожидает `SCM_invoke`.

Упражнение 10.2

Добавьте глобальным переменным флажок, показывающий их инициализированность. Его начальное значение устанавливается в функции `objectify-free-global-reference`.

```
(define-class Global-Variable Variable (initialized?))

(define (objectify-free-global-reference name r)
  (let ((v (make-Global-Variable name #f)))
    (insert-global! v r)
    (make-Global-Reference v) ) )
```

Затем встройте анализ глобальных переменных в компилятор. Он будет выполняться обходчиком кода с помощью обобщённой функции `inian!`.

```
(define (compile->C e out)
  (set! g.current '())
  (let ((prg (extract-things!
              (lift! (initialization-analyze! (Sexp->object e))) )))
    (gather-temporaries! (closurize-main! prg))
    (generate-C-program out e prg) ) )

(define (initialization-analyze! e)
  (call/cc (lambda (exit)
             (inian! e (lambda () (exit 'finished))) )) )

(define-generic (inian! (e) exit)
  (update-walk! inian! e exit) )
```

Задачей этой функции будет выявить все глобальные переменные, которые гарантированно получили значение до того, как это значение кому-то потребовалось. Сложность выполнения данного анализа зависит от желаемого уровня общности. Мы выберем простой путь и определим все глобальные переменные, которые всегда инициализируются.

```
(define-method (inian! (e Global-Assignment) exit)
  (call-next-method)
  (let ((gv (Global-Assignment-variable e)))
    (set-Global-Variable-initialized! gv #t)
    (inian-warning "Surely initialized variable" gv)
    e ) )
```

```
(define-method (inian! (e Global-Reference) exit)
  (let ((gv (Global-Reference-variable e)))
    (cond ((Predefined-Variable? gv) e)
          ((Global-Variable-initialized? gv) e)
          (else (inian-error "Surely uninitialized variable" gv)
                (exit) ) ) ) )

(define-method (inian! (e Alternative) exit)
  (inian! (Alternative-condition e) exit)
  (exit) )

(define-method (inian! (e Application) exit)
  (call-next-method)
  (exit) )

(define-method (inian! (e Function) exit) e)
```

Анализатор проходит по коду, находит все присваивания глобальным переменным и останавливается, когда программа становится слишком сложной; то есть когда он встречает ветвление или вызов функции. Кстати, `lambda`-формы не являются «слишком сложным кодом», так как они всегда безошибочно вычисляются за конечное время и не трогают глобальные переменные.

Упражнение 11.1

Предикат `Object?` можно улучшить, добавив в векторы, которыми представляются объекты, ещё одно поле, хранящее уникальную метку. Соответственно, также потребуется изменить аллокаторы, чтобы они заполняли это поле во всех создаваемых объектах. (И не забыть добавить его в примитивные классы, которые определяются вручную.)

```
(define *starting-offset* 2)
(define meroonet-tag (cons 'meroonet 'tag))

(define (Object? o)
  (and (vector? o)
       (>= (vector-length o) *starting-offset*)
       (eq? (vector-ref o 1) meroonet-tag) ) )
```

При таком подходе предикат `Object?` будет реже ошибаться, но ценой этого является некоторая потеря быстродействия. Однако, его всё равно можно обмануть, ведь не мешает пользователю извлечь метку из любого объекта с помощью `vector-ref` и вставить её в какой-нибудь другой вектор.

Упражнение 11.2

Так как это обобщённая функция, то её можно специализировать для конкретных классов. Универсальная реализация слишком уж неэффективно расходует память:

```
(define-generic (clone (o))
  (list->vector (vector->list o)) )
```

Упражнение 11.3

Определите новый класс классов: метакласс `CountingClass`, у которого есть поле для подсчёта создаваемых объектов.

```
(define-class CountingClass Class (counter))
```

К счастью, MEROONET написана так, что для её расширения не требуется изменять половину существующих определений. Новый метакласс можно определить как-то так:

```
(define-meroonet-macro (define-CountingClass name super-name
                                             own-fields )
  (let ((class (register-CountingClass name super-name own-fields)))
    (generate-related-names class) ) )

(define (register-CountingClass name super-name own-fields)
  (CountingClass-initialize! (allocate-CountingClass)
                             name
                             (->Class super-name)
                             own-fields ) )
```

Однако более правильным решением будет расширить синтаксис формы `define-class` так, чтобы она принимала тип создаваемого класса (по умолчанию `Class`). При этом потребуется сделать некоторые функции обобщёнными:

```
(define-generic (generate-related-names (class)))

(define-method (generate-related-names (class Class))
  (Class-generate-related-names class) )

(define-generic (initialize! (o) . args))

(define-method (initialize! (o Class) . args)
  (apply Class-initialize o args) )

(define-method (initialize! (o CountingClass) . args)
  (set-CountingClass-counter! class 0)
  (call-next-method) )
```

Обновлять значение поля `counter` будут, конечно же, аллокаторы нового метакласса:

```
(define-method (generate-related-names (class CountingClass))
  (let* ((cname      (symbol-concatenate (Class-name class) '-class))
         (alloc-name (symbol-concatenate 'allocate- (Class-name class)))
         (make-name  (symbol-concatenate 'make- (Class-name class))) )
```

```
    `(begin ,(call-next-method)
            (set! ,alloc-name                ; аллокатор
                  (let ((old ,alloc-name))
                    (lambda sizes
                      (set-CountingClass-counter! ,cname
                       (+ 1 (CountingClass-counter ,cname)) )
                      (apply old sizes) ) ) )
            (set! ,make-name                 ; конструктор
                  (let ((old ,make-name))
                    (lambda args
                      (set-CountingClass-counter! ,cname
                       (+ 1 (CountingClass-counter ,cname)) )
                      (apply old args) ) ) ) ) ) )
```

В качестве заключения рассмотрим пример использования данного метакласса:

```
(define-CountingClass CountedPoint Object (x y))

(unless (and (= 0 (CountingClass-counter CountedPoint-class))
             (allocate-CountedPoint)
             (= 1 (CountingClass-counter CountedPoint-class))
             (make-CountedPoint 11 22)
             (= 2 (CountingClass-counter CountedPoint-class)) )
  ;; не выполнится, если всё в порядке
  (meroonet-error "Failed test on CountedPoint") )
```

Упражнение 11.4

Определите метакласс `ReflectiveClass`, обладающий дополнительными полями: `predicate`, `allocator` и `maker`. Затем измените определение генератора сопутствующих функций, чтобы он заполнял эти поля при создании экземпляра класса. Аналогичные действия необходимо выполнить для классов полей (наследников `Field`).

```
(define-class ReflectiveClass Class (predicate allocator maker))

(define-method (generate-related-names (class ReflectiveClass))
  (let ((cname      (symbol-concatenate (Class-name class) '-class))
        (pred-name  (symbol-concatenate (Class-name) '?))
        (alloc-name (symbol-concatenate 'allocate- (Class-name class)))
        (make-name  (symbol-concatenate 'make- (Class-name class))) )
    `(begin ,(call-next-method)
            (set-ReflectiveClass-predicate! ,cname ,pred-name)
            (set-ReflectiveClass-allocator! ,cname ,alloc-name)
            (set-ReflectiveClass-maker!     ,cname ,make-name) ) ) )
```

Упражнение 11.5

Главная сложность здесь в том, как узнать, существует ли уже обобщённая функция или нет. В Scheme нельзя определить, существует или нет глобальная переменная, поэтому придётся искать имя функции в списке `*generics*`.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym)) (c (gensym)))
       `(begin
          (unless (->Generic ',(car call))
            (define-generic ,call) )
          (register-method
           ',(car call)
           (lambda (,g ,c)
             (lambda ,(flat-variables variables)
               (define (call-next-method)
                 ((if (Class-superclass ,c)
                      (vector-ref (Generic-dispatch-table ,g)
                                  (Class-number (Class-superclass ,c)) )
                      (Generic-default ,g) )
                  . ,(flat-variables variables) ) )
               . ,body ) )
           ',(cadr discriminant)
           ',(cdr call) ) ) ) ) ) )
```

Упражнение 11.6

Просто добавьте в определение каждого метода пару локальных функций `call-next-method` и `next-method?`. Несомненно, было бы лучше сделать так, чтобы эти функции создавались только тогда, когда они действительно используются, но это реализовать сложнее.

```
(define-meroonet-macro (define-method call . body)
  (parse-variable-specifications
   (cdr call)
   (lambda (discriminant variables)
     (let ((g (gensym)) (c (gensym)))
       `(register-method
         ',(car call)
         (lambda (,g ,c)
           (lambda ,(flat-variables variables)
             ,@(generate-next-method-functions g c variables)
             . ,body ) )
         ',(cadr discriminant)
         ',(cdr call) ) ) ) ) )
```

Функция `next-method?` похожа на `call-next-method`, но она только ищет суперметод, не вызывая его.

```
(define (generate-next-method-functions g c variables)
  (let ((get-next-method (gensym)))
    `((define (,get-next-method)
        (if (Class-superclass ,c)
            (vector-ref (Generic-dispatch-table ,g)
                        (Class-number (Class-superclass ,c)) )
            (Generic-default ,g) ) )
      (define (call-next-method)
        ((,get-next-method) . ,(flat-variables variables)) )
      (define (next-method?)
        (not (eq? (,get-next-method) (Generic-default ,g))) ) ) ) )
```

Библиография

[85M85] Macscheme Reference Manual / Semantic Microsystems. — Sausalito, California, 1985.

[All78] *John Allen.* Anatomy of Lisp. — McGraw-Hill, 1978. — 464 p. — (Computer Science Series).

[ALQ95] *S. Anglade, J.-J. Lacrampe, and C. Queinnec.* Semantics of Combinations in Scheme // ACM SIGPLAN Lisp Pointers. — 1994. Vol. 7, iss. 4. — P. 15–20.

[App87] *Andrew Appel.* Garbage Collection Can Be Faster Than Stack Allocation // Information Processing Letters. — 1987. Vol. 25, iss. 4. — P. 275–279.

[App92a] *Andrew Appel.* Compiling with Continuations. — Cambridge Press, 1992. — 272 p.

[App92b] Dylan, an Object-Oriented Dynamic Language / Apple Computer Eastern Research & Technology. — Apple Computer, Inc., 1992.

[AR88] *N. Adams and J. Rees.* Object-Oriented Programming in Scheme // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 277–288.

[AS85] *Harold Abelson and Gerald Jay Sussman with Julie Sussman.* Structure and Interpretation of Computer Programs. — MIT Press, 1985. — 400 p.

[AS85′] *Харольд Абельсон и Джеральд Джей Сассмен.* Структура и интерпретация компьютерных программ. — Добросвет, 2010. — 608 с.

[AS94] An Empirical and Analitical Study of Stack vs Heap Cost for Languages with Closures : technical report : CS-TR-450-94 / Princeton University, Dept. of Computer Science ; A. Appel and Zh. Shao. — 1994.

[Att95] *Giuseppe Attardi.* The Embeddable Common Lisp // ACM SIGPLAN Lisp Pointers. — 1995. Vol. 8, iss. 1. — P. 30–41.

[Bak] *Henry G. Baker.* CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. // ACM SIGPLAN Notices. — 1995. Vol. 30, iss. 9. — P. 17–20.

[Bak78] *Henry G. Baker.* List Processing in Real Time on a Serial Computer // Communications of the ACM. — 1978. Vol. 21, iss. 4. — P. 280–294.

[Bak92a] *Henry G. Baker.* The Buried Binding and Dead Binding Problems of Lisp 1.5 // ACM SIGPLAN Lisp Pointers. — 1992. Vol. 5, iss. 2. — P. 11–19.

[Bak92b] *Henry G. Baker.* Inlining Semantics for Subroutines Which Are Recursive // ACM SIGPLAN Lisp Pointers. — 1992. Vol. 27, iss. 12. — P. 39–46.

[Bak92c] *Henry G. Baker.* Metacircular Semantics for Common Lisp Special Forms // ACM SIGPLAN Lisp Pointers. — 1992. Vol. 5, iss. 4. — P. 11–12.

[Bak93] *Henry G. Baker.* Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same // ACM SIGPLAN OOPS Messenger. — 1993. Vol. 4, iss. 4. — P. 2–27.

[Bar84] *Hendrik Pieter Barendregt.* The Lambda Calculus, Its Syntax and Semantics. — North Holland, 1984. — 621 p. — (Studies in Logic and the Foundations of Mathematics ; Vol. 103).

[Bar84′] *Хенк Барендрегт.* Лямбда-исчисление. Его синтаксис и семантика. — М.: Мир, 1985. — 606 с.

[Bar89] Scheme→C, a Portable Scheme-to-C Compiler : research report : WRL-89-1 / DEC Western Research Laboratory ; Joel F. Bartlett. — 1989.

[Baw88] *Andrew Bawden.* Reification without Evaluation // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 342–349.

[BC87] *J.-P. Briot and P. Cointe.* A Uniform Model for Object-Oriented Languages Using the Class Abstraction // Intern. Joint Conf. on Artificial Intelligence. Vol. 1. — 1987. — P. 40–43.

[BCSJ86] *J.-P. Briot, P. Cointe et E. Saint-James.* Reécriture et récursion dans une fermeture etude dans un Lisp à liason superficielle et application aux objets // BIGRE+Globule. — 1986. Vol. 48. — P. 90–100.

[BDG+88] Common Lisp Object System Specification / D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, et al. // ACM SIGPLAN Notices. — 1988. Vol. 23. — 142 p. — Spec. iss.

[Bel73] *James R. Bell.* Threaded Code // Communications of the ACM. — 1973. Vol. 16, iss. 6. — P. 370–372.

[Bet91] *David M. Betz.* XSCHEME: An Object-Oriented Scheme. — P.O. Box 144, Peterborough, NH 03458 (USA), 1991. — Version 0.28

[BG94] *Henri Elle Bal and Dick Grune.* Programming Language Essentials. — Addison-Wesley, 1994. — 288 p.

[BHY88] *A. Bloss, P. Hundak, and J. Young.* Code Optimizations for Lazy Evaluation // Lisp and Symbolic Computation. — 1988. Vol. 1, iss. 2. — P. 147–164.

[BJ86] *D. H. Bartley and J. C. Jensen.* The Implementation of PC Scheme // Proc. of the 1986 ACM Conf. on LISP and Functional Programming. — 1986. — P. 86–93.

[BKK+86] CommonLoops: Merging Lisp and Object-Oriented Programming / D. G. Bobrow, K. Kahn, G. Kiczales, et al. // OOPSLA'86 Conf. Proc. on Object-Oriented Programming Systems, Languages, and Applications. — 1986. — P. 17–29.

[BM82] A Mechanical Proof of the Unsolvability of the Halting Problem : technical report : ICSCA-CMP-28 / Institute for Computing Science and Computer Applications ; R. S. Boyer and J S. Moore. — 1982.

[BR88] *A. Bawden and J. Rees.* Syntactic Closures // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 86–95.

[BW88] *H.-J. Boehm and M. Weiser.* Garbage Collection in an Uncooperative Environment // Software—Practice & Experience. — 1988. Vol. 18, iss. 9. — P. 807–820.

[Car93] An implementation of $F_{<:}$: research report : 97 / DEC Systems Research Center ; Luca Cardelli. — 1993.

[Car94] *George J. Carrette.* SIOD: Scheme in One Defun. — 1994.

[Cay83] *Michel Cayrol.* Le langage Lisp. — Cepadues Editions, 1983. — 143 p.

[CC77] *P. Cousot and R. Cousot.* Abstract Interpretation: A Unified Lattice Model for Static Analisys of Programs by Construction or Approximation of Fixpoints // Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. — 1977. — P. 238–252.

[CDD+91] Le-Lisp version 15.24, Le manuel de référence / J. Chailloux, M. Devin, F. Dupont, et al. ; INRIA. — 1991.

[CEK+89] Recommended C Style and Coding Standards / L. W. Cannon, R. A. Elliot, L. W. Kirchhoff, et al. — 1989.

[CG77] *D. W. Clark and C. C. Green.* An Empirical Study of List Structure in LISP // Communications of the ACM. — 1977. Vol. 20, iss. 2. — P. 78–87.

[CH94] *W. D. Clinger and L. T. Hansen.* Lambda, the Ultimate Label, or A Simple Optimizing Compiler for Scheme // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 128–139.

[Cha80] *Jérôme Chailloux.* Le modèle VLISP : description, implémentation et évaluation : thèse de troisième cycle / Unversité Paris-VIII. — 1980.

[Cha96] *Gregory J. Chaitin.* The Limits of Mathematics // Journal of Universal Computer Science. — 1996. Vol. 2, iss. 5. — P. 270–305.

[CHO88] *W. D. Clinger, A. H. Hartheimer, and E. M. Ost.* Implementation Strategies for Continuations // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 124–131.

[Chr95] *Christophe.* La Famille Fenouillard. — Armand Colin, 1895.

[Cla79] *Douglas W. Clark.* Measurements of Dynamic List Structure Use in Lisp // IEEE Trans. on Software Engineering. — 1979. Vol. 5, iss. 1. — P. 51–59.

[Cli84] *William D. Clinger.* The Scheme 311 compiler: An Exercise in Denotational Semantics // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 356–364.

[Coi87] *Pierre Cointe.* The ObjVlisp Kernel: A Reflexive Architecture to Define a Uniform Object-Oriented System // Workshop on Meta-Level Architecture and Reflection. — 1987. — Was not published.

[Com84] *Douglas Comer.* Operating System Design: The XINU Approach. — Prentice-Hall, 1984. — 496 p.

[CR91a] *W. Clinger and J. Rees.* Macros that Work // Proc. of the 18th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1991. — P. 155–162.

[CR91b] Revised^4 Report on the Algorithmic Language Scheme / ed. by. W. Clinger and J. Rees // ACM SIGPLAN Lisp Pointers. — 1991. Vol. 4, iss. 3. — 55 p.

[Cur89] *Pavel Curtis.* (Algorithms) // ACM SIGPLAN Lisp Pointers. — 1989. Vol. 3, iss. 1. — P. 48–61.

[Dan87] *Olivier Danvy.* Memory Allocation and Higher-Order Functions // Proc. of the ACM SIGPLAN'87 Symp. on Interpreters and Interpretive Techniques. — 1987. — P. 241–252.

[Del89] *Vincent Delacour.* Picolo expresso // BIGRE+Globule. — 1989. Iss. 65. — P. 30–42.

[Deu80] *L Peter Deutsch.* ByteLisp and Its Alto Implemenatation // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 231–242.

[Deu89] Génération automatique d'interpreteurs et compilation à partir de spécifications dénotationnelles : rapport de DEA-LAP : LITP-RXF 89-17 / Laboratoire d'Informatique Théorique et Programmation ; Alain Deutsch. — 1989.

[Dev85] Le portage du système Le-Lisp : rapport technique : 50 / INRIA-Rocquencourt ; Matthieu Devin. — 1985.

[DF90] *O. Danvy and A. Filinski.* Abstracting Control // Proc. of the 1990 ACM Conf. on Lisp and Functional Programming. — 1990. — P. 151–160.

[DFH86] *R. K. Dybvig, D. P. Friedman, and C. T. Haynes.* Expansion-Passing Style: Beyond Conventional Macros // Proc. of the 1986 ACM Conf. on LISP and Functional Programming. — 1986. — P. 143–150.

[DFH88] *R. K. Dybvig, D. P. Friedman, and C. T. Haynes.* Expansion-Passing Style: A General Macro Mechanism // Lisp and Symbolic Computation. — 1988. Vol. 1, iss. 1. — P. 53–76.

[DH92] *O. Danvy and J. Hatcliff.* Thunks (continued) // Proc. of the 2nd Intern. Workshop on Static Analysis. — 1992. — P. 3–11. — See also technical report CIS-92-28, Kansas State University.

[DHB93] *R. K. Dybvig, R. G. Hieb, and C. T. Bruggerman.* Syntactic Abstaction in Scheme // Lisp and Symbolic Computation. — 1993. Vol. 5, iss. 4. — P. 295–326.

[DHHM92] Monotonic Conflict Resolution Mechanisms for Inheritance / R. Ducournau, M. Habib, M. Huchard, et al. // OOPSLA'92 Conf. Proc. on Object-Oriented Programming Systems, Languages, and Applications. — 1992. — P. 16–24.

[Dil88] *Antoni Diller.* Compiling Functional Languages. — John Wiley & Sons, 1988. — 322 p.

[DM88] A Blond Primer : rapport : DIKU 88/21 / University of Copenhagen ; O. Danvy and K. Malmkjær. — 1988.

[DM95] *Antione Dumesnil de Maricourt.* Macro-expansion en Lisp, sémantique et rélisation : thèse d'université / Université Paris-VII. — 1995.

[DPS94a] *H. Davis, P. Parquier, and N. Séniak.* Sweet Harmony: The Talk/C++ Connection // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 121–127.

[DPS94b] *H. Davis, P. Parquier, and N. Séniak.* Talking About Modules and Delivery // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 113–120.

[dR87] *Jim des Rivières.* Control-Related Meta-Level Facilities in Lisp // Workshop on Meta-Level Architecture and Reflection. — 1987. — P. 101–110.

[dRS84] *J. des Rivières and B. C. Smith.* The Implemenatation of Procedurally Reflective Languages // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 331–347.

[Dyb87] *R. Kent Dybvig.* The SCHEME Programming Language. — Prentice-Hall, 1987. — 242 p.

[ES70] *J. Earley and H. Sturgis.* A Formalism for Translator Interaction // Communications of the ACM. — 1970. Vol. 13, iss. 10. — P. 607–617.

[Fel88] *Matthias Felleisen.* The Theory and Practice of First-Class Prompts // Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1988. — P. 180–190.

[Fel90] *Matthias Felleisen.* On the Expressive Power of Programming Languages // ESOP'90 Selected Papers from the Symp. on the 3rd European Symposium on Programming. — 1990. — P. 134–151.

[FF87] *M. Felleisen and D. P. Friedman.* A Reduction Semantics for Imperative Higher-Order Languages // Parallel Architectures and Languages Europe. Vol. II. — 1987. — P. 206-223.

[FF89] *M. Felleisen and D. P. Friedman.* A Syntactic Theory of Sequential State // Theoretical Computer Science. — 1989. Vol. 69, iss. 3. — P. 243–287.

[FFDM87] Beyond Continuations : technical report : TR216 / Indiana University, Computer Science Dept. ; M. Felleisen, D. P. Friedman, B. Duba, et al.. — 1987.

[FH89] The Revised Report on the Syntactic Theories of Sequential Control and State : technical report : 100-89 / Rice University ; M. Felleisen and R. Hieb. — 1989.

[FL87] *M. Feeley and G. LaPalme.* Using Closures for Code Generation // Computer Languages. — 1987. Vol. 12, iss. 1. — P. 47–66.

[FM90] *M. Feeley and J. S. Miller.* A Parallel Virtual Machine for Efficient Scheme Compilation // Proc. of the 1990 ACM Conf. on Lisp and Functional Programming. — 1990. — P. 119–130.

[FW76] *D. P. Friedman and D. S. Wise.* CONS Should Not Evaluate Its Arguments // Proc. of the 3rd Intern. Colloquium on Automata, Languages and Programming. — 1976. — P. 257–284.

[FW84] *D. P. Friedman and M. Wand.* Reification: Reflection Without Metaphysics // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 348–355.

[FWFD88] Abstract Continuations: A Mathematical Semantics for Handling Functional Jumps / M. Felleisen, M. Wand, D. P. Friedman, et al. // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 52–62.

[FWH92] *Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes.* Essentials of Programming Languages. — MIT Press and McGraw-Hill, 1992. — 536 p.

[Gab88] *Richard P. Gabriel.* The Why of Y // ACM SIGPLAN Lisp Pointers. — 1988. Vol. 2, iss. 2. — P. 15–25.

[GBM82] *M. L. Griss, E. Benson, and G. Q. Maguire Jr.* PSL: A Portable LISP System // Proc. of the 1982 ACM Symp. on LISP and Functional Programming. — 1982. — P. 88–97.

[Gor75] Operational Reasoning and Denotational Semantics : research report : STAN-CS-506 / Stanford University, Computer Science Dept. ; Michael Gordon. — 1975.

[Gor88] *Michael Gordon.* Programming Language Theory and its Implemenatation. — Prentice-Hall, 1988. — 255 p. — (International Series in Computer Science).

[GP88] *R. P. Gabriel and K. M. Pitman.* Technical Issues of Separation in Function Cells and Value Cells // Lisp and Symbolic Computation. — 1988. Vol. 1, iss. 1. — P. 81–101.

[GR83] *Adèle Goldberg and David Robson.* Smalltalk-80: The Language and its Implemenatation. — Addison-Wesley, 1983. — 714 p.

[Gra93] *Paul Graham.* On Lisp: Advanced Techniques for Common Lisp. — Prentice-Hall, 1993. — 413 p.

[Gre77] *Patrick Greussay.* Contribution à la définiton interprètative et à l'implémentation des Lambda-langages : thèse d'état / Unversité Paris-VI. — 1977.

[Gud93] Representing Type Information in Dynamically Typed Languages : technical report : 93-27 / University of Arizona, Dept. of Computer Science ; David Gudeman. — 1993.

[Han90] *Chris Hanson.* Efficient Stack Allocation for Tail-Recursive Languages // Proc. of the 1990 ACM Conf. on Lisp and Functional Programming. — 1990. — P. 106–118.

[HD90] *R. Hieb and R. K. Dybvig.* Continuations and Concurrency // Proc. of the 2nd ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming. — 1990. — P. 128–136.

[HDB90] *R. Hieb, R. K. Dybvig, and C. Bruggerman.* Representing Control in the Presence of First-Class Continuations // Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation. — 1990. — P. 66–77.

[Hen80] *Peter Henderson.* Functional Programming: Application and Implementation. — Prentice-Hall, 1980. — 350 p. — (International Series in Computer Science).

[Hen80′] *Питер Хендерсон.* Функциональное программирование. Применение и реализация. — М.: Мир, 1983. — 349 с. — (Математическое обеспечение ЭВМ).

[Hen92a] *Fritz Henglein.* Global Tagging Optimizations by Type Inference // Proc. of the 1992 ACM Conf. on Lisp and Functional Programming. — 1992. — P. 205–215.

[Hen92b] *Wade Henessey.* WCL: Delivering Efficient Common Lisp Applications Under Un⋆x // Proc. of the 1992 ACM Conf. on Lisp and Functional Programming. — 1992. — P. 260-269.

[HFW84] *C. T. Haynes, D. P. Friedman, and M. Wand.* Continuations and Coroutines // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 293-298.

[Hof93] *Ulrich Hoffmann.* Using C as Target Code for Translating High-Level Programming Languages // Journal of C Language Translation. — 1993. Vol. 5, iss. 2. — P. 70–90.

[Hon93] *P. Joseph Hong.* Threaded Code Designs for Forth Interpreters // ACM SIGFORTH Newsletter. — 1993. Vol. 4, iss. 2. — P. 11–16.

[HS75] *C. Hewitt and B. Smith.* Towards a Programming Apprentice // IEEE Trans. on Software Engineering. — 1975. Vol. 1, iss. 1. — P. 26–45.

[HS91] *Samuel P. Harbison and Guy Lewis Steele Jr.* C: A Reference Manual. — 3rd edition. — Prentice-Hall, 1991. — 392 p.

[IEE91] IEEE 1178-1990. IEEE Standard for the Scheme Programming Language. — Institute of Electrical and Electronic Engineers, 1991.

[ILO94] Ilog Talk Reference Manual / ILOG. — 1994.

[IM89] *Takayasu Ito and Manabu Matsui.* A Parallel Lisp Language PaiLisp and Its Kernel Specification // Proc. of the US/Japan Workshop on Parallel Lisp: Languages and Systems. — 1989. — P. 58–100.

[IMY92] *Yuuju Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa.* RbCL: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel // Proc. of Intern. Workshop on New Models for Software Architecture. — 1992. — P. 24–35.

[ISO90] ISO/IEC 9899:1990. Programming Languages – C. — International Organization for Standardization, 1990.

[ISO94] ISO/IEC JTC1/SC22/WG16. Programming Language ISLISP. — International Organization for Standardization, 1994.

[ITW86] *Takayasu Ito, Takashi Tamura, and Shinichi Wada.* Theoretical Comparisons of Interpreted/Compiled Executions of Lisp on Sequential and Parallel Machine Models // Proc. of the IFIP 10th World Computer Congress. — 1986. — P. 349–354.

[Jaf94] *Aubrey Jaffer.* Reference Manual for SCM. — 1994.

[JF92] *S. Jefferson and D. P. Friedman.* A Simple Reflective Interpreter // Proc. of Intern. Workshop on New Models for Software Architecture. — 1992. — P. 48–58.

[JGS93] *Neil D. Jones, Carsten K. Gomard, and Peter Sestoft.* Partial Evaluation and Automatic Program Generation. — Prentice-Hall, 1993. — 400 p. — (International Series in Computer Science). — With chapters by L. O. Andersen and T. Mogensen.

[Kah87] *Gilles Kahn.* Natural Semantics // Proc. of the 4th Annual Symp. on Theoretical Aspects of Computer Science. — 1987. — P. 22–39.

[Kam90] *Samuel N. Kamin.* Programming Languages: An Interpreter-Based Approach. — Addison-Wesley, 1990. — 640 p.

[KCR98] Revised^5 Report on the Algorithmic Language Scheme / ed. by. R. Kelsey, W. Clinger, and J. Rees // ACM SIGPLAN Notices. — 1998. Vol. 33, iss. 9. — 50 p.

[KdRB92] *Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow.* The Art of the Metaobject Protocol. — MIT Press, 1992. — 345 p.

[Kes88] *Robert R. Kessler.* Lisp, Objects, and Symbolic Programming. — Scott Foresman & Co, 1988. — 644 p.

[KFFD86] Hygienic Macro Expansion / E. E. Kohlbecker, D. P. Friedman, M. Felleisen, et al. // Proc. of the 1986 ACM Conf. on LISP and Functional Programming. — 1986. — P. 151–161.

[KH89] *R. Kelsey and P. Hudak.* Realistic Compilation by Program Transformation // Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1989. — P. 281–292.

[Knu84] *Donald Ervin Knuth.* The TEXbook. — Addison-Wesley, 1984. — 496 p.

[Knu84′] *Дональд Эрвин Кнут.* Всё про TEX. — Вильямс, 2003. — 560 с. — (Компьютеры и вёрстка).

[KR78] *Brian W. Kernighan and Dennis M. Ritchie.* The C Programming Language. — Prentice-Hall, 1978. — 228 p.

[KR90] *G. Kiczales and L. Rodriguez.* Efficient Method Dispatch in PCL // Proc. of the 1990 ACM Conf. on Lisp and Functional Programming. — 1990. — P. 99–105.

[KW90] *M. Katz and D. Weise.* Continuing into the Future: On the Interaction of Futures and First-Class Continuations // Proc. of the 1990 ACM Conf. on Lisp and Functional Programming. — 1990. — P. 176–184.

[Lak80] *Fred H. Lakin.* Computing with Text-Graphical Forms // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 100–106.

[Lan65] *Peter J. Landin.* Correspondence between ALGOL 60 and Church's Lambda-notation // Communications of the ACM. — 1965. Vol. 8. — P. 89–101 and 158–165.

[Leb05] *Maurice Leblanc.* 813. — Éditions Pierre Lafitte, 1905.

[LF88] *H. Lieberman and C. Fry.* Common EVAL // ACM SIGPLAN Lisp Pointers. — 1988. Vol. 2, iss. 1. — P. 23–33.

[LF93] *S.-D. Lee and D. P. Friedman.* Quasi-Static Scoping: Sharing Variable Bindings Across Multiple Lexical Scopes // Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1993. — P. 479–492.

[Lie87] *Henry Lieberman.* Reversible Object-Oriented Interpreters // Proc. of the ECOOP'87 European Conf. on Object-Oriented Programming. — 1987. — P. 11–19.

[LLSt93] GNU Emacs Lisp Reference Manual / B. Lewis, D. LaLiberte, R. Stallman, and GNU Manual Group ; Free Software Foundation. — 2nd edition. — 1993.

[LP86] *K. J. Lang and B. A. Pearlmutter.* Oaklisp: An Object-Oriented Scheme with First Class Types // OOPSLA'86 Conf. Proc. on Object-Oriented Programming Systems, Languages, and Applications. — 1986. — P. 30–37.

[LP88] *K. J. Lang and B. A. Pearlmutter.* Oaklisp: An Object-Oriented Dialect of Scheme // Lisp and Symbolic Computation. — 1988. Vol. 1, iss. 1. — P. 39–51.

[LW93] *Xavier Leroy et Pierre Weis.* Manuel de référence du langage Caml. — InterÉditions, 1993.

[MAE+62] LISP 1.5 Programmer's Manual / John McCarthy, Paul W. Abrahams, Daniel J. Edwards, et al. — MIT Press, 1962. — 112 p.

[Man74] *Zohar Manna.* Mathematical Theory of Computation. — McGraw-Hill, 1974. — 448 p. — (Computer Science Series).

[Mas86] *Ian A. Mason.* The Semantics of Destructive LISP. — CSLI, 1986. — 300 p. — (Lecture notes).

[Mat92] *Luis Mateu.* An Efficient Implemenatation of Coroutines // Proc. of the IWMM'92 Intern. Workshop on Memory Management. — 1992. — P. 230–247.

[MB93] *Luis Mateu-Brule.* Stratégies avancées pour la gestion de blocs de contrôle : thèse de doctorat d'université / Unversité Paris-VI. — 1993.

[McC60] *John McCarthy.* Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I // Communications of the ACM. — 1960. Vol. 3, iss. 1. — P. 184–195.

[McC78a] *John McCarthy.* History of LISP // ACM SIGPLAN Notices. — 1978. Vol. 13, iss. 8. — P. 217–223.

[McC78b] *John McCarthy.* A micro-manual for LISP – Not the whole truth // ACM SIGPLAN Notices. — 1978. Vol. 13, iss. 8. — P. 215–216.

[McD93] The Relatedness and Comparative Utility of Various Approaches to Operational Semantics : technical report : MS-CIS-93-16 / University of Pennsylvaina, CIS Dept. ; Raymond McDowell. — 1993.

[MNC+89] Les langages à objets / Gérald Masini, Amedeo Napoli, Dominique Colnet, et al. — InterÉditions, 1989.

[MNC+89′] Object-Oriented Languages / Gérald Masini, Amedeo Napoli, Dominique Colnet, et al. — Academic Press, 1991. — 512 p.

[Mor92] *Luc Moreau.* An Operational Semantics for a Parallel Functional Language with Continuations // Proc. of the 4th Intern. Conf. on Parallel Architectures and Languages Europe. — 1992. — P. 415–420.

[Mor94] *Luc Moreau.* Sound Evaluation of Parallel Functional Programs with First-Class Continuations : PhD thesis / University of Liège. — 1994.

[Mos70] *Joel Moses.* The Function of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem // ACM SIGSAM Bulletin. — 1970. Iss. 15. — P. 13–27.

[Moz87] *Вольфганг Амадей Моцарт.* Дон Жуан [опера]. — 1787.

[MP80] *S. S. Muchnick and U. F. Pleban.* A Semantic Comparison of LISP and Scheme // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 56–65.

[MQ94] *L. Moreau and C. Queinnec.* Partial Continuations as the Difference of Continuations – A Duumvirate of Control Operators // Proc. of the 6th Intern. Symp. on Programming Language Implementation and Logic Programming. — 1994. — P. 182–197.

[MR91] *J. S. Miller and G. J. Rozas.* Free Variables and First-Class Environments // Lisp and Symbolic Computation. — 1991. Vol. 4, iss. 2. — P. 107–141.

[MS80] *F. L. Morris and J. S. Schwarz.* Computing Cyclic List Structures // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 144–153.

[Mul92] *Robert Muller.* M-Lisp: A Representation-Independent Dialect of Lisp with Reduction Semantics // ACM Trans. on Programming Languages and Systems. — 1992. Vol. 14, iss. 4. — P. 589–615.

[Nei84] *Eugen Neidl.* Étude des relations avec l'interprète dans la compilation de Lisp : Thèse de troisième cycle / Unversité Paris-VI. — 1984.

[Nor72] *Eric J. Norman.* 1100 LISP Reference Manual. — University of Wisconsin, 1972.

[NQ89] Identifier Semantics: A Matter of References : technical report : LIX RR 89 02 / Laboratoire d'Informatique de l'École Polytechnique ; G. Nuyens and C. Queinnec. — 1989. — P. 67–80.

[PE92] The EuLisp Definition / ed. by. J. Padget and G. Nuyens. — 1992.

[Per79] *Jean-François Perrot.* Lisp et λ-calcul // Actes de la 6e école de printemps d'informatique théorique. — 1979. — P. 277–301.

[Pit80] *Kent Pitman.* Special Forms in LISP // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 179–187.

[PJ87] *Simon L. Peyton-Jones.* The Implemenatation of Functional Programming Languages. — Prentice-Hall, 1987. — 500 p. — (International Series in Computer Science).

[PNB93] *J. Padget, G. Nuyens, and H. Bretthauer.* An Overview of EuLisp // Lisp and Symbolic Computation. — 1993. Vol. 6, iss. 1/2. — P. 9–98.

[QC88] *C. Queinnec and P. Cointe.* An Open-Ended Data Representation Model for EuLisp // Proc. of the 1988 ACM Conf. on Lisp and Functional Programming. — 1988. — P. 298–308.

[QD93] *C. Queinnec and D. DeRoure.* Design of a concurrent and distributed language // Proc. of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications. — 1993. — P. 234–259.

[QD96] *C. Queinnec and D. DeRoure.* Sharing Code through First-Class Environments // Proc. of the 1996 ACM SIGPLAN Intern. Conf. on Functional Programming. — 1996. — P. 251–261.

[QG92] *C. Queinnec and J.-M. Geffroy.* Partial Evaluation Applied to Symbolic Pattern Matching with Intelligent Backtrack // Actes WSA'92 Workshop on Static Analysis (Bordeaux). — 1992. — P. 109–117.

[QP90] A Deterministic Model for Modules and Macros : technical report : 90-36 / University of Bath, Bath Computing Group ; C. Queinnec and J. Padget. — 1990.

[QP91a] *C. Queinnec and J. Padget.* A Proposal for a Modular Lisp with Macros and Dynamic Evaluation // Journées de Travail sur l'Analyse Statique en Programmation Équationnelle, Fonctionnelle et Logique. — 1991. — P. 1–8.

[QP91b] *C. Queinnec and J. Padget.* Modules, Macros and Lisp // The 11th Intern. Conf. of the Chilean Computer Science Society. — 1991. — P. 111–123.

[QS91] *C. Queinnec and B. Serpette.* A Dynamic Extent Control Operator for Partial Continuations // Proc. of the 18th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1991. — P. 174–184.

[Que82] *Christian Queinnec.* Langage d'un autre type : LISP. — Eyrolles, 1982. — 184 p.

[Que89] Lisp – Almost a whole Truth! : technical report : LIX RR 89 03 / Laboratoire d'Informatique de l'École Polytechnique ; Christian Queinnec. — 1989. — P. 79–105.

[Que90a] *Christian Queinnec.* A Framework for Data Aggregates // Actes des JFLA 90 – Journées Francophones des Langages Applicatifs. — 1990. — P. 21–32.

[Que90b] *Christian Queinnec.* Le filtrage : une application de (et pour) Lisp. — InterÉditions, 1990. — 201 p.

[Que90c] *Christian Queinnec.* PolyScheme: A Semantics for a Concurrent Scheme // High Performance and Parallel Computing in Lisp. — 1990.

[Que92a] *Christian Queinnec.* Compiling Syntactically Recursive Programs // ACM SIGPLAN Lisp Pointers. — 1992. Vol. 5, iss. 4. — P. 2–10.

[Que92b] *Christian Queinnec.* A Concurrent and Distributed Extension of Scheme // Proc. of the 4th Intern. Conf. on Parallel Architectures and Languages Europe. — 1992. — P. 431–446.

[Que93a] *Christian Queinnec.* Continuation Conscious Compilation // ACM SIGPLAN Lisp Pointers. — 1993. Vol. 6, iss. 1. — P. 2–14.

[Que93b] *Christian Queinnec.* Designing Meroon V3 // ECOOP'93 Workshop on Object-Oriented Programming in Lisp: Languages and Applications. — 1993.

[Que93c] *Christian Queinnec.* A Library of High-Level Control Operators // ACM SIGPLAN Lisp Pointers. — 1993. Vol. 6, iss. 4. — P. 11–26.

[Que93d] Literate Programming from Scheme to TEX : research report : LIX RR 93 05 / Laboratoire d'Informatique de l'École Polytechnique ; Christian Queinnec. — P. 79–105. — 1993.

[Que94] *Christian Queinnec.* Locality, Causality and Continuations // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 91–102.

[Que95] *Christian Queinnec.* DMeroon. Overview of a Distributed Class-based Causality-coherent Data Model // Proc. of the Intern. Workshop PSLS'95 on Parallel Symbolic Languages and Systems. — 1995. — P. 297–309.

[R3R86] Revised^3 Report on the Algorithmic Language Scheme / ed. by. W. Clinger and J. Rees // ACM SIGPLAN Notices. — 1986. Vol. 21, iss. 12. — P. 37–79.

[RA82] *J. A. Rees and N. I. Adams.* T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool // Proc. of the 1982 ACM Symp. on LISP and Functional Programming. — 1982. — P. 114–122.

[RAM84] *J. A. Rees, N. I. Adams, and J. R. Meehan.* The T Manual / Yale University, Computer Science Dept. — 4th edition. — 1984.

[Ray91] *Eric Raymond.* The New Hacker's Dictionary. — MIT Press, 1991.

[Rey72] *John C. Reynolds.* Definitional Interpreters for Higher-Order Programming Languages // Proc. of the ACM Annual Conference. — 1972. — P. 717–740.

[Rib69] *Daniel Ribbens.* Programmation non numérique : LISP 1.5. — Dunod, 1969. — (Monographies d'Informatique).

[RM92] *J. R. Rose and H. Muller.* Integrating the Scheme and C Languages // Proc. of the 1992 ACM Conf. on Lisp and Functional Programming. — 1992. — P. 247–259.

[Row80] *William Rowan.* A LISP Compiler Producing Compact Code // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 216–222.

[Roz92] *Guillermo Juan Rozas.* Taming the Y Operator // Proc. of the 1992 ACM Conf. on Lisp and Functional Programming. — 1992. — P. 226–234.

[Sam79] *Hanan Sammet.* Deep and Shallow Binding: The Assignment Operation // Computer Languages. — 1979. Vol. 4. — P. 187–198.

[Sch86] *David Schmidt.* Denotational Semantics: A Methodology for Language Development. — Allyn and Bacon, 1986. — 348 p.

[Sco76] *Dana Scott.* Data Types as Lattices // SIAM Journal on Computing. — 1976. Vol. 5, iss. 3. — P. 522–587.

[Sén89] *Nitsan Séniak.* Compilation de Scheme par spécialisation explicite // BIGRE+Globule. — 1989. Iss. 65. — P. 160–170.

[Sén91] *Nitsan Séniak.* Théorie et practique de Sqil, un langage intermédiare pour la compilation des langages fonctionnels : thèse de doctorat d'université / Unversité Paris-VI. — 1991.

[Ser93] *Manuel Serrano.* De l'utilisation des analyses de flot de contrôle dans la compilation des langages fonctionnels // Actes des journées du GDR de Programmation. — 1993. — P. 226–234. — See also PLILP'94 proc., P. 447–448.

[Ser94] *Manuel Serrano.* Bigloo's User Manual. — 1994.

[SF89] *George Springer and Daniel P. Friedman.* Scheme and the Art of Programming. — MIT Press, 1989. — 600 p.

[SF92] *A. Sabry and M. Felleisen.* Reasoning About Programs in Continuation-Passing Style // Proc. of the 1992 ACM Conf. on Lisp and Functional Programming. — 1992. — P. 288–298.

[SG93] *G. L. Steele Jr and R. P. Gabriel.* The Evolution of Lisp // ACM SIGPLAN Notices. — 1993. Vol. 28, iss. 3. — P. 231–270.

[Shi91] *Olin Shivers.* Data-flow Analysis and Type Recovery in Scheme // Topics in Advanced Language Implemenatation / ed. by. Peter Lee. — MIT Press, 1991.

[SJ87] *Emmanuel Saint-James.* De la méta-récursivité comme outil d'implémentation : thèse d'état / Unversité Paris-VI. — 1987.

[SJ93] *Emmanuel Saint-James.* La programmation applicative (de LISP à la machine en passant par le lambda-calcul). — Hermès, 1993. — 411 p.

[Sla61] *James R. Slagle.* A Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, Symbolic Automatic Integration (SAINT) : PhD thesis / MIT, Dept. of Mathematics. — 1961.

[Spi90] *Éric Spir.* Gestion dynamique de la mémoire dans les langages de programmation, application à Lisp. — InterÉditions, 1990. — 156 p. — (Science informatique).

[SS75] Scheme: An Interpreter for Extended Lambda Calculus : AI Memo : 349 / Massachusetts Institute of Technology ; G. J. Sussman and G. L. Steele Jr. — 1975.

[SS78a] The Art of the Interpreter or The Modularity Complex (Parts Zero, One, and Two) : AI Memo : 453 / Massachusetts Institute of Technology ; G. L. Steele Jr and G. J. Sussman. — 1978.

[SS78b] The Revised Report on SCHEME, a Dialect of LISP : AI Memo : 452 / Massachusetts Institute of Technology ; G. L. Steele Jr and G. J. Sussman. — 1978.

[SS80] *G. L. Steele Jr and G. J. Sussman.* The Dreams of Lifetime: A Lazy Variable Extent Mechanism // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 163–172.

[Ste78] RABBIT: A Compiler for SCHEME : AI Memo : 474 / Massachusetts Institute of Technology ; Guy Lewis Steele Jr. — 1978.

[Ste84] *Guy Lewis Steele Jr.* Common Lisp. The Language. — Digital Press, 1984. — 465 p.

[Ste90] *Guy Lewis Steele Jr.* Common Lisp. The Language. — 2nd edition. — Digital Press, 1990. — 1029 p.

[Sto77] *Joseph E. Stoy.* Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. — MIT Press, 1977. — 414 p.

[Str86] *Bjarne Stroustrup.* The C++ Programming Language. — Addison-Wesley, 1986. — 326 p.

[SW94] *M. Serrano and P. Weis.* 1 + 1 = 1: An Optimizing Caml Compiler // Record of the 1994 ACM SIGPLAN Workshop on ML and Its Applications. — 1994. — P. 101–111.

[Tak88] *Masato Takeichi.* Lambda-Hoisting: A Transformation Technique for Fully Lazy Evaluation of Functional Programs // New Generation Computing. — 1988. Vol. 5, iss. 4. — P. 377–391.

[Tei74] *Warren Teitelman.* InterLISP Reference Manual / Xerox Palo Alto Research Center — 1974.

[Tei76] *Warren Teitelman.* CLisp: Coversational LISP // IEEE Trans. on Computers. — 1976. Vol. 25, iss. 4. — P. 354–357.

[VH94] *J. Vitek and R. N. Horspool.* Taming Message Passing: Efficient Method Look-Up for Dynamically Typed Languages // Proc. of the 8th European Conf. on Object-Oriented Programming. — 1994. — P. 432–449.

[Wad88] *Philip Wadler.* Deforestation: Transforming Programs to Eliminate Trees // Proc. of the 2nd European Symposium on Programming. — 1988. — P. 344–358.

[Wan80a] *Mitchell Wand.* Continuation-Based Multiprocessing // Proc. of the 1980 ACM LISP Conference. — 1980. — P. 19–28.

[Wan80b] *Mitchell Wand.* Continuation-Based Program Transformation Strategies // Journal of the ACM. — 1980. Vol. 27, iss. 1. — P. 164–180.

[Wan84] *Mitchell Wand.* A Semantic Prototyping System // Proc. of the 1984 SIGPLAN Symp. on Compiler Construction. — 1984. — P. 213–221.

[Wan86] *M. Wand and D. P. Friedman.* The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower // Proc. of the 1986 ACM Conf. on LISP and Functional Programming. — 1986. — P. 298–307.

[Wat93] *Richard C. Waters.* MACROEXPAND-ALL: An Example of a Simple Lisp Code Walker // ACM SIGPLAN Lisp Pointers. — 1993. Vol. 6, iss. 1. — P. 25–32.

[WC94] *A. K. Wright and R. Cartwright.* A Practical Soft Type System for Scheme // Proc. of the 1984 ACM Symp. on LISP and Functional Programming. — 1984. — P. 250–262.

[WH89] *Patrick H. Winston and Berthold K. Horn.* LISP. — 3rd edition. — Addison-Wesley, 1989. — 611 p.

[Wil92] *Paul R. Wilson.* Uniprocessor Garbage Collection Techniques // Proc. of the IWMM'92 Intern. Workshop on Memory Management. — 1992. — P. 1–42.

[WL93] *Pierre Weis et Xavier Leroy.* Le langage Caml. — InterÉditions, 1993.

[WS91] *Larry Wall and Randal L. Schwartz.* Programming Perl. — O'Reilly Media, 1991. — 482 p.

[WS94] *M. Wand and P. Steckler.* Selective and Lightweight Closure Conversion // Proc. of the 21th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1994. — P. 435–445.

[YH85] *Taiichi Yuasa and Masami Hagiya.* Kyoto Common Lisp Report / Kyoto University. — 1985.

[YS92] Proceedings of International Workshop on New Models for Software Architecture: Reflection and Meta-Level Architecture / ed. by. Akinori Yonezawa and Brian C. Smith. — Tokyo: RISE, 1992.

Предметный указатель

C

D

E

F

G

I

N

O

P

Q

R

S

T

U

V

W

Y

Z

A

Б

В

Г

Д

К

Л

О

П

Т

У

Ф

Х