Christian Queinnec

Lisp in Small Pieces

CAMBRIDGE

more information – www.cambridge.org/9780521562478

Lisp in Small Pieces

Lisp in Small Pieces

Christian Queinnec
Ecole Polytechnique

Translated by Kathleen Callaway

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York NY 10011–4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org
Information on this title: www.cambridge.org/9780521562478

First published in French as *Les Langages Lisp* by Interéditions 1994

First published in English 1996
First paperback edition 2003

Translated by
Kathleen Callaway, 9 allée des Bois du Stade, 33700 Merignac, France

A catalogue record for this book is available from the British Library

ISBN 0 521 56247 3 hardback
ISBN 0 521 54566 8 paperback

Table of Contents

To the Reader

EVEN though the literature about Lisp is abundant and already accessible to the reading public, nevertheless, this book still fills a need. The logical substratum where Lisp and Scheme are founded demand that modern users must read programs that use (and even abuse) advanced technology, that is, higher-order functions, objects, continuations, and so forth. Tomorrow's concepts will be built on these bases, so not knowing them blocks your path to the future.

To explain these entities, their origin, their variations, this book will go into great detail. Folklore tells us that even if a Lisp user knows the value of every construction in use, he or she generally does not know its cost. This work also intends to fill that mythical hole with an in-depth study of the semantics and implementation of various features of Lisp, made more solid by more than thirty years of history.

Lisp is an enjoyable language in which numerous fundamental and non-trivial problems can be studied simply. Along with ML, which is strongly typed and suffers few side effects, Lisp is the most representative of the applicative languages. The concepts that illustrate this class of languages absolutely must be mastered by students and computer scientists of today and tomorrow. Based on the idea of "function," an idea that has matured over several centuries of mathematical research, applicative languages are omnipresent in computing; they appear in various forms, such as the composition of UN⋆X byte streams, the extension language for the EMACS editor, as well as other scripting languages. If you fail to recognize these models, you will misunderstand how to combine their primitive elements and thus limit yourself to writing programs painfully, word by word, without a real architecture.

Audience

This book is for a wide, if specialized audience:

- to graduate students and advanced undergraduates who are studying the implementation of languages, whether applicative or not, whether interpreted, compiled, or both.
- to programmers in Lisp or Scheme who want to understand more clearly the costs and nuances of constructions they're using so they can enlarge their expertise and produce more efficient, more portable programs.

- to the lovers of applicative languages everywhere; in this book, they'll find many reasons for satisfying reflection on their favorite language.

Philosophy

This book was developed in courses offered in two areas: in the graduate research program (DEA ITCP: Diplôme d'Études Approfondies en Informatique Théorique, Calcul et Programmation) at the University of Pierre and Marie Curie of Paris VI; some chapters are also taught at the École Polytechnique.

A book like this would normally follow an introductory course about an applicative language, such as Lisp, Scheme, or ML, since such a course typically ends with a description of the language itself. The aim of this book is to cover, in the widest possible scope, the semantics and implementation of interpreters and compilers for applicative languages. In practical terms, it presents no less than twelve interpreters and two compilers (one into byte-code and the other into the C programming language) without neglecting an object-oriented system (one derived from the popular MEROON). In contrast to many books that omit some of the essential phenomena in the family of Lisp dialects, this one treats such important topics as reflection, introspection, dynamic evaluation, and, of course, macros.

This book was inspired partly by two earlier works: *Anatomy of Lisp* [All78], which surveyed the implementation of Lisp in the seventies, and *Operating System Design: the Xinu Approach* [Com84], which gave all the necessary code without hiding any details on how an operating system works and thereby gained the reader's complete confidence.

In the same spirit, we want to produce a precise (rather than concise) book where the central theme is the semantics of applicative languages generally and of Scheme in particular. By surveying many implementations that explore widely divergent aspects, we'll explain in complete detail how any such system is built. Most of the schisms that split the community of applicative languages will be analyzed, taken apart, implemented, and compared, revealing all the implementation details. We'll "tell all" so that you, the reader, will never be stumped for lack of information, and standing on such solid ground, you'll be able to experiment with these concepts yourself.

Incidentally, all the programs in this book can be picked up, intact, electronically (details on page xix).

Structure

This book is organized into two parts. The first takes off from the implementation of a naive Lisp interpreter and progresses toward the semantics of Scheme. The line of development in this part is motivated by our need to be more specific, so we successively refine and redefine a series of name spaces (Lisp_1, Lisp_2, and so forth), the idea of continuations (and multiple associated control forms), assignment, and writing in data structures. As we slowly augment the language that we're defining, we'll see that we inevitably pare back its defining language so that it is reduced to

Chapter	Signature
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 sr) r k)`
	`((meaning e sr tail?) k)`
	`((meaning e sr tail?))`
7	`(run (meaning e sr tail?))`
10	`(->C (meaning e sr))`

Figure 1 Approximate signatures of interpreters and compilers

a kind of λ-calculus. We then convert the description we've gotten this way into its denotational equivalent.

More than six years of teaching experience convinced us that this approach of making the language more and more precise not only initiates the reader gradually into authentic language-research, but it is also a good introduction to denotational semantics, a topic that we really can't afford to leap over.

The second part of the book goes in the other direction. Starting from denotational semantics and searching for efficiency, we'll broach the topic of fast interpretation (by pretreating static parts), and then we'll implement that preconditioning (by precompilation) for a byte-code compiler. This part clearly separates program preparation from program execution and thus handles a number of topics: dynamic evaluation (`eval`); reflective aspects (first class environments, auto-interpretable interpretation, reflective tower of interpreters); and the semantics of macros. Then we introduce a second compiler, one compiling to the C programming language.

We'll close the book with the implementation of an object-oriented system, where objects make it possible to define the implementation of certain interpreters and compilers more precisely.

Good teaching demands a certain amount of repetition. In that context, the number of interpreters that we examine, all deliberately written in different styles—naive, object-oriented, closure-based, denotational, etc.—cover the essential techniques used to implement applicative languages. They should also make you think about the differences among them. Recognizing these differences, as they are sketched in Figure 1, will give you an intimate knowledge of a language and its implementation. Lisp is not just one of these implementations; it is, in fact, a *family* of dialects, each one made up of its own particular mix of the characteristics we'll be looking at.

In general, the chapters are more or less independent units of about forty pages or so; each is accompanied by exercises, and the solutions to those exercises are found at the end of the book. The bibliography contains not only historically important references, so you can see the evolution of Lisp since 1960, but also

references to current, on-going research.

Prerequisites

Though we hope this book is both entertaining and informative, it may not necessarily be easy to read. There are subjects treated here that can be appreciated only if you make an effort proportional to their innate difficulty. To harken back to something like the language of courtly love in medieval France, there are certain objects of our affection that reveal their beauty and charm only when we make a chivalrous but determined assault on their defences; they remain impregnable if we don't lay seige to the fortress of their inherent complexity.

In that respect, the study of programming languages is a discipline that demands the mastery of tools, such as the λ-calculus and denotational semantics. While the design of this book will gradually take you from one topic to another in an orderly and logical way, it can't eliminate all effort on your part.

You'll need certain prerequisite knowledge about Lisp or Scheme; in particular, you'll need to know roughly thirty basic functions to get started and to understand recursive programs without undue labor. This book has adopted Scheme as the presentation language; (there's a summary of it, beginning on page xviii) and it's been extended with an object layer, known as MEROON. That extension will come into play when we want to consider problems of representation and implementation.

All the programs have been tested and actually run successfully in Scheme. For readers that have assimilated this book, those programs will pose no problem whatsoever to port!

Thanks and Acknowledgments

I must thank the organizations that procured the hardware (Apple Mac SE30 then Sony News 3260) and the means that enabled me to write this book: the École Polytechnique, the Institut National de Recherche en Informatique et Automatique (INRIA-Rocquencourt, the national institute for research in computing and automation at Rocquencourt), and Greco-PRC de Programmation du Centre National de la Recherche Scientifique (CNRS, the national center for scientific research, special group for coordinated research on computer science).

I also want to thank those who actually participated in the creation of this book by all the means available to them. I owe particular thanks to Sophie Anglade, Josy Baron, Kathleen Callaway, Jérôme Chailloux, Jean-Marie Geffroy, Christian Jullien, Jean-Jacques Lacrampe, Michel Lemaître, Luc Moreau, Jean-François Perrot, Daniel Ribbens, Bernard Serpette, Manuel Serrano, Pierre Weis, as well as my muse, Claire N.

Of course, any errors that still remain in the text are surely my own.

Notation

Extracts from programs appear in **this type face**, no doubt making you think unavoidably of an old-fashioned typewriter. At the same time, certain parts will appear in *italic* to draw attention to variations within this context.

The sign → indicates the relation "has this for its value" while the sign ≡ indicates equivalence, that is, "has the same value as." When we evaluate a form in detail, we'll use a vertical bar to indicate the environment in which the expression must be considered. Here's an example illustrating these conventions in notation:

```
(let ((a (+ b 1)))
  (let ((f (lambda () a)))
    (foo (f) a) ) )|        ;the value of foo is the function for creating dotted
                   |b→ 3  ;pairs that is, the value of the global variable cons.
                   |foo≡ cons
≡ (let ((f (lambda () a))) (foo (f) a))|
                                       |a→ 4
                                       |b→ 3
                                       |foo≡ cons
                                       |f≡ (lambda () a)|
                                                         |a→ 4
≡ (foo (f) a)|
             |a→ 4
             |b→ 3
             |foo≡ cons
             |f≡ (lambda () a)|
                               |a→ 4
→ (4 . 4)
```

We'll use a few functions that are non-standard in Scheme, such as **gensym** that creates symbols guaranteed to be new, that is, different from any symbol seen before. In Chapter 10, we'll also use **format** and **pp** to display or "pretty-print." These functions exist in most implementations of Lisp or Scheme.

Certain expressions make sense only in the context of a particular dialect, such as COMMON LISP, Dylan, EULISP, IS-Lisp, Le-Lisp[1], Scheme, etc. In such a case, the name of the dialect appears next to the example, like this:

```
(defdynamic fooncall                                        IS-Lisp
  (lambda (one :rest others)
    (funcall one others) ) )
```

To make it easier for you to get around in this book, we'll use this sign [**see p.**] to indicate a cross-reference to another page. When we suggest variations detailed in the exercises, we'll also use that sign, like this [**see Ex.**]. You'll also find a complete index of the function definitions that we mention. [**see p. 495**]

1. Le-Lisp is a trademark of INRIA.

Short Summary of Scheme

There are excellent books for learning Scheme, such as [AS85, Dyb87, SF89]. For reference, the standard document is the *Revised revised revised revised Report on Scheme*, informally known as R^{4}RS.

This summary merely outlines the important characteristics of that dialect, that is, the characteristics that we'll be using later to dissect the dialect as we lead you to a better understanding of it.

Scheme lets you handle symbols, characters, character strings, lists, numbers, Boolean values, vectors, ports, and functions (or procedures in Scheme parlance).

Each of those data types has its own associated predicate: `symbol?`, `char?`, `string?`, `pair?`, `number?`, `boolean?`, `vector?`, and `procedure?`.

There are also the corresponding selectors and modifiers, where appropriate, such as: `string-ref`, `string-set!`, `vector-ref`, and `vector-set!`.

For lists, there are: `car`, `cdr`, `set-car!`, and `set-cdr!`.

The selectors, `car` and `cdr`, can be composed (and pronounced), so, for example, to designate the second term in a list, we use `cadr` and pronounce it something like *kadder*.

These values can be implicitly named and created simply by mentioning them, as we do with symbols and identifiers. For characters, we prefix them by `#\` as in `#\Z` or `#\space`. We enclose character strings within quotation marks (that is, ") and lists within parentheses (that is, ()). We use numbers are they are. We can also make use of Boolean values, namely, `#t` and `#f`. For vectors, we use this syntax: `#(do re mi)`, for example. Such values can be constructed dynamically with `cons`, `list`, `string`, `make-string`, `vector`, and `make-vector`. They can also be converted from one to another, by using `string->symbol` and `int->char`.

We manage input and output by means of these functions: `read`, of course, reads an expression; `display` shows an expression; `newline` goes to the next line.

Programs are represented by Scheme values known as *forms*.

The form `begin` lets you group forms to evaluate them sequentially; for example, `(begin (display 1) (display 2) (newline))`.

There are many conditional forms. The simplest is the form *if—then—else*—conventionally written in Scheme this way: `(if` *condition then otherwise*`)`. To handle choices that entail more than two options, Scheme offers `cond` and `case`. The form `cond` contains a group of *clauses* beginning with a Boolean form and ending by a series of forms; one by one the Boolean forms of the clauses are evaluated until one returns true (or more precisely, not false, that is, not `#f`); the forms that follow the Boolean form that succeeded will then be evaluated, and their result becomes the value of the entire `cond` form. Here's an example of the form `cond` where you can see the default behavior of the keyword `else`.

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

The form `case` has a form as its first parameter, and that parameter provides a key that we'll look for in all the clauses that follow; each of those clauses specifies

which key or keys will set it off. Once an appropriate key is found, the associated forms will be evaluated and their result will become the result of the entire **case** form. Here's how we would convert the preceding example using **cond** into one using **case**.

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

Functions are defined by a **lambda** form. Just after the keyword **lambda**, you'll find the variables of the function, followed by the expressions that indicate how to calculate the function. These variables can be modified by assignment, indicated by **set!**. Literal constants are introduced by **quote**. The forms **let**, **let***, and **letrec** introduce local variables; the initial value of such a local variable may be calculated in various ways.

With the form **define**, you can define named values of any kind. We'll exploit the internal writing facilities that **define** forms provide, as well as the non-essential syntax where the name of the function to define is indicated by the way it's called. Here is an example of what we mean.

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

That example could also be rewritten without inessential syntax, like this:

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

That example completes our brief summary of Scheme.

Programs and More Information

The programs (both interpreted and compiled) that appear in this book, the object system, and the associated tests are all available on-line by anonymous **ftp** at:

(IP 128.93.2.54) **ftp.inria.fr:INRIA/Projects/icsla/Books/LiSP.tar.gz**

At the same site, you'll also find articles about Scheme and other implementations of Scheme.

The electronic mail address of the author of this book is:

Christian.Queinnec@polytechnique.fr

Recommended Reading

Since we assume that you already know Scheme, we'll refer to the standard reference [AS85, SF89].

To gain even greater advantage from this book, you might also want to prepare yourself with other reference manuals, such as COMMON LISP [Ste90], Dylan [App92b], EULISP[PE92], IS-Lisp [ISO94], Le-Lisp [CDD+91], OakLisp [LP88], Scheme [CR91b], T [RAM84] and, Talk [ILO94].

Then, for a wider perspective about programming languages in general, you might want to consult [BG94].

1
The Basics of Interpretation

THIS chapter introduces a basic interpreter that will serve as the foundation for most of this book. Deliberately simple, it's more closely related to Scheme than to Lisp, so we'll be able to explain Lisp in terms of Scheme that way. In this preliminary chapter, we'll broach a number of topics in succession: the articulations of this interpreter; the well known pair of functions, `eval` and `apply`; the qualities expected in environments and in functions. In short, we'll start various explorations here to pursue in later chapters, hoping that the intrepid reader will not be frightened away by the gaping abyss on either side of the trail.

The interpreter and its variations are written in native Scheme without any particular linguistic restrictions.

Literature about Lisp rarely resists that narcissistic pleasure of describing Lisp in Lisp. This habit began with the first reference manual for Lisp 1.5 [MAE+62] and has been widely imitated ever since. We'll mention only the following examples of that practice: (There are many others.) [Rib69], [Gre77], [Que82], [Cay83], [Cha80], [SJ93], [Rey72], [Gor75], [SS75], [All78], [McC78b], [Lak80], [Hen80], [BM82], [Cli84], [FW84], [dRS84], [AS85], [R3R86], [Mas86], [Dyb87], [WH88], [Kes88], [LF88], [Dil88], [Kam90].

Those evaluators are quite varied, both in the languages that they define and in what they use to do so, but most of all in the goals they pursue. The evaluator defined in [Lak80], for example, shows how graphic objects and concepts can emerge naturally from Lisp, while the evaluator in [BM82] focuses on the size of evaluation.

The language used for the *definition* is important as well. If assignment and surgical tools (such as `set-car!`, `set-cdr!`, and so forth) are allowed in the definition language, they enrich it and thus minimize the size (in number of lines) of descriptions; indeed, with them, we can precisely simulate the language *being defined* in terms that remind us of the lowest level machine instructions. Conversely, the description uses more concepts. Restricting the definition language in that way complicates our task, but lowers the risk of semantic divergence. Even if the size of the description grows, the language being defined will be more precise and, to that degree, better understood.

Figure 1.1 shows a few representative interpreters in terms of the complexity

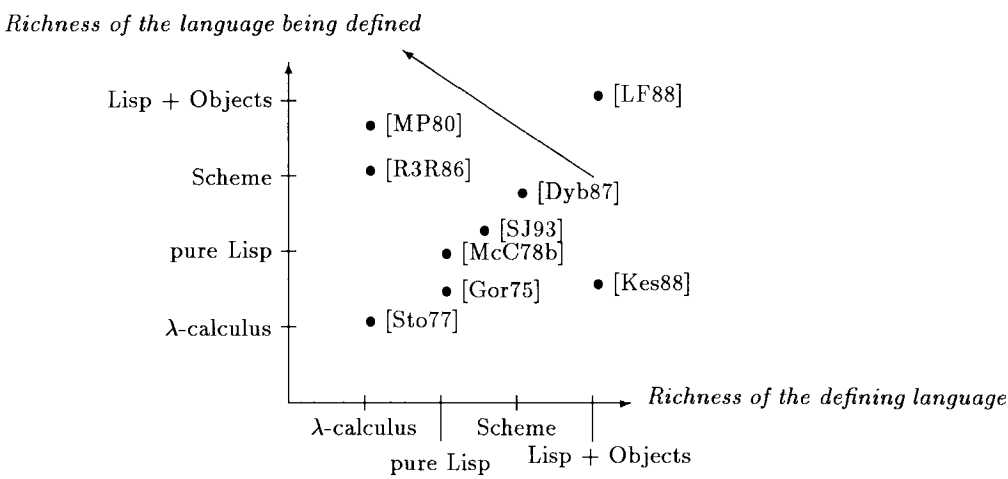


Figure 1.1

of their definition language (along the x-axis) and the complexity of the language being defined (along the y-axis). The knowledge progression shows up very well in that graph: more and more complicated problems are attacked by more and more restricted means. This book corresponds to the vector taking off from a very rich version of Lisp to implement Scheme in order to arrive at the λ-calculus implementing a very rich Lisp.

1.1 Evaluation

The most essential part of a Lisp interpreter is concentrated in a single function around which more useful functions are organized. That function, known as **eval**, takes a program as its argument and returns the value as output. The presence of an explicit evaluator is a characteristic trait of Lisp, not an accident, but actually the result of deliberate design.

We say that a language is *universal* if it is as powerful as a Turing machine. Since a Turing machine is fairly rudimentary (with its mobile read-write head and its memory composed of binary cells), it's not too difficult to design a language that powerful; indeed, it's probably more difficult to design a useful language that is not universal.

Church's thesis says that any function that can be computed can be written in any universal language. A Lisp system can be compared to a function taking programs as input and returning their value as output. The very existence of such systems proves that they are computable: thus a Lisp system can be written in a universal language. Consequently, the function **eval** itself can be written in Lisp, and more generally, the behavior of Fortran can be described in Fortran, and so forth.

What makes Lisp unique—and thus what makes an explication of **eval** non-trivial—is its reasonable size, normally from one to twenty pages, depending on the

level of detail.[1] This property is the result of a significant effort in design to make the language more regular, to suppress special cases, and above all to establish a syntax that's both simple and abstract.

Many interesting properties result from the existence of `eval` and from the fact that it can be defined in Lisp itself.

- You can learn Lisp by reading a reference manual (one that explains functions thematically) or by studying the `eval` function itself. The difficulty with that second approach is that you have to know Lisp in order to read the definition of `eval`—though knowing Lisp, of course, is the *result* we're hoping for, rather than the *prerequisite*. In fact, it's sufficient for you to know only the subset of Lisp used by `eval`. The language that defines `eval` is a pared-down one in the sense that it procures only the essence of the language, reduced to special forms and primitive functions.

  It's an undeniable advantage of Lisp that it brings you these two intertwined approaches for learning it.

- The fact that the definition of `eval` is available in Lisp means that the programming environment is part of the language, too, and costs little. By programming environment, we mean such things as a tracer, a debugger, or even a reversible evaluator [Lie87]. In practice, writing these tools to control evaluation is just a matter of elaborating the code for `eval`, for example, to print function calls, to store intermediate results, to ask the end-user whether he or she wants to go on with the evaluation, and so forth.

  For a long time, these qualities have insured that Lisp offers a superior programming environment. Even today, the fact that `eval` can be defined in Lisp means that it's easy to experiment with new models of implementation and debugging.

- Finally, `eval` itself is a programming tool. This tool is controversial since it implies that an application written in Lisp and using `eval` must include an entire interpreter or compiler, but more seriously, it must give up the possibility of many optimizations. In other words, using `eval` is not without consequences. In certain cases, its use is justified, notably when Lisp serves as the definition and the implementation of an incremental programming language.

  Even apart from that important cost, the semantics of `eval` is not clear, a fact that justifies its being separated from the official definition of Scheme in [CR91b]. [see p. 271]

1.2 Basic Evaluator

Within a program, we distinguish *free variables* from *bound variables*. A variable is free as long as no binding form (such as `lambda`, `let`, and so forth) qualifies it; otherwise, we say that a variable is bound. As the term indicates, a free variable is unbound by any constraint; its value could be anything. Consequently, in order

1. In this chapter, we define a Lisp of about 150 lines.

to know the value of a fragment of a program containing free variables, we must know the values of those free variables themselves. The data structure associating variables and values is known as an *environment.* The function **evaluate**[2] is thus binary; it takes a program accompanied by an environment and returns a value.

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

1.3 Evaluating Atoms

An important characteristic of Lisp is that programs are represented by expressions of the language. However, since any representation assumes a degree of encoding, we have to explain more about how programs are represented. The principal conventions of representation are that a variable is represented by a symbol (its name) and that a functional application is represented by a list where the first term of the list represents the function to apply and the other terms represent arguments submitted to that function.

Like any other compiler, **evaluate** begins its work by syntactically analyzing the expression to evaluate in order to deduce what it represents. In that sense, the title of this section is inappropriate since this section does not literally involve evaluating atoms but rather evaluating programs where the representation is atomic. It's important, in this context, to distinguish the program from its representation (or, the message from its medium, if you will). The function **evaluate** works on the representation; from the representation, it deduces the expected intention; finally, it executes what's requested.

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

If an expression is not a list, perhaps it's a symbol or actual data, such as a number or a character string. When the expression is a symbol, the expression represents a *variable* and its value is the one attributed by the environment.

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

The function **lookup** (which we'll explain later on page 13) knows how to find the value of a variable in an environment. Here's the signature of **lookup**:

(**lookup** *variable environment)* → *value*

2. There is a possibility of confusion here. We've already mentioned the evaluator defined by the function **eval**; it's widely present in any implementation of Scheme, even if not standardized; it's often unary—accepting only one argument. To avoid confusion, we'll call the function **eval** that we are defining by the name **evaluate** and the associated function **apply** by the name **invoke**. These new names will also make your life easier if you want to experiment with these programs.

In consequence, an implicit conversion takes place between a symbol and a variable. If we were more meticulous about how we write, then in place of `(lookup exp env)`, we should have written:

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

That more scrupulous way of writing emphasizes that the symbol—the value of `exp`—must be changed into a variable. It also underscores the fact that the function `symbol->variable`[3] is not at all an identity; rather, it converts a syntactic entity (the symbol) into a semantic entity (the variable). In practice, then, a variable is nothing other than an imaginary object to which the language and the programmer attach a certain sense but which, for practical reasons, is handled only by means of its representation. The representation was chosen for its convenience: `symbol->variable` works like the identity because Lisp exploits the idea of a symbol as one of its basic types. In fact, other representations could have been adopted; for example, a variable could have appeared in the form of a group of characters, prefixed by a dollar sign. In that case, the conversion function `symbol->variable` would have been less simple.

If a variable were an imaginary concept, the function `lookup` would not know how to accept it as a first argument, since `lookup` knows how to work only on tangible objects. For that reason, once again, we have to encode the variable in a representation, this time, a key, to enable `lookup` to find its value in the environment. A precise way of writing it would thus be:

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

But the natural laziness of Lisp-users inclines them to use the symbol of the same name as the key associated with a variable. In that context, then, `variable->key` is merely the inverse of `symbol->variable` and the composition of those two functions is simply the identity.

When an expression is atomic (that is, when it does not involve a dotted pair) and when that expression is not a symbol, we have the habit of considering it as the representation of a constant that is its own value. This idempotence is known as the *autoquote* facility. An autoquoted object does not need to be quoted, and it is its own value. See [Cha94] for an example.

Here again, this choice is not obvious for several reasons. Not all atomic objects naturally denote themselves. The value of the character string `"a?b:c"` might be to call the C compiler for this string, then to execute the resulting program, and to insert the results back in Lisp.

Other types of objects (functions, for example) seem stubbornly resistant to the idea of evaluation. Consider the variable `car`, one that we all know the utility of; its value is the function that extracts the left child from a pair; but that function *car* itself—what is its value? Evaluating a function usually proves to be an error that should have been detected and prevented earlier.

Another example of a problematic value is the empty list `()`. From the way it is written, it might suggest that it is an empty application, that is, a functional application without any arguments where we've even forgotten to mention the

3. Personally, I don't like names formed like this x`->`y to indicate a conversion because this order makes it difficult to understand compositions; for example, (`y->z(x->y...)`) is less straightforward than (`z<-y(y<-x...)`). In contrast, x`->`y is much easier to read than y`<-`x. You can see here one of the many difficulties that language designers come up against.

function. That syntax is forbidden in Scheme and consequently is not defined as having a value.

For those kinds of reasons, we have to analyze expressions very carefully, and we should *autoquote* only those data that deserve it, namely, numbers, characters, and strings of characters. [see p. 7] We could thus write:

```
(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)) )
      ... ) )
```

In that fragment of code, you can see the first case of possible errors. Most Lisp systems have their own exception mechanism; it, too, is difficult to write in portable code. In an error situation, we could call `wrong`[4] with a character string as its first argument. That character string would describe the kind of error, and the following arguments could be the objects explaining the reason for the anomaly. We should mention, however, that more rudimentary systems send out cryptic messages, like `Bus error: core dump` when errors occur. Others stop the current computation and return to the basic interaction loop. Still others associate an exception handler with a computation, and that exception handler catches the object representing the error or exception and decides how to behave from there. [see p. 255] Some systems even offer exception handlers that are quasi-expert systems themselves, analyzing the error and the corresponding code to offer the end-user choices about appropriate corrections. In short, there is wide variation in this area.

1.4 Evaluating Forms

Every language has a number of syntactic forms that are "untouchable": they cannot be redefined adequately, and they must not be tampered with. In Lisp, such a form is known as a *special form.* It is represented by a list where the first term is a particular symbol belonging to the set of *special operators.*[5]

A dialect of Lisp is characterized by its set of special forms and by its library of primitive functions (those functions that cannot be written in the language itself and that have profound semantic repercussions, as, for example, `call/cc` in Scheme).

In some respects, Lisp is simply an ordinary version of applied λ-calculus, augmented by a set of special forms. However, the special genius of a given Lisp is expressed in just this set. Scheme has chosen to minimize the number of special operators (`quote`, `if`, `set!`, and `lambda`). In contrast, COMMON LISP (CLtL2

4. Notice that we did not say "the function `wrong`." We'll see more about error recovery on page 255.

5. We will follow the usual lax habit of considering a special operator, say `if`, as being a "special form" although `if` is not even a form. Scheme treats them as "syntactic keywords" whereas COMMON LISP recognizes them with the `special-form-p` predicate.

[Ste90]) has more than thirty or so, thus circumscribing the number of cases where it's possible to generate highly efficient code.

Because special forms are coded as they are, their syntactic analysis is simple: it is based on the first term of each such form, so one **case** statement suffices. When a special form does not begin with a keyword, we say that is it is a *functional application* or more simply an application. For the moment, we're looking at only a small subset of the general special forms: **quote**, **if**, **begin**, **set!**, and **lambda**. (Later chapters introduce new, more specialized forms.)

```
(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) )) ) ) )
```

In order to lighten that definition, we've reduced the syntactic analysis to its minimum, and we haven't bothered to verify whether the quotations are well formed, whether **if** is really ternary[6] (accepting three parameters) each time, and so forth. We'll assume that the programs that we're analyzing are syntactically correct.

1.4.1 Quoting

The special form **quote** makes it possible to introduce a value that, without its quotation, would have been confused with a legal expression. The decision to represent a program as a value of the language makes it necessary, when we want to speak about a particular value, to find a means for discriminating between data and programs that usurp the same space. A different syntactic choice could have avoided that problem. For example, M-expressions, originally planned in [McC60] as the normal syntax for programs written in Lisp, would have eliminated this particular problem, but they would have forbidden macros—a marvelously useful tool for extending syntax; however, M-expressions disappeared rapidly [McC78a]. The special form **quote** is consequently the principal discriminator between program and data.

6. As a form, **if** is not necessarily ternary. That is, it does not have to have three parameters. Scheme and COMMON LISP, for example, support both binary and ternary **if**, whereas EULISP and IS-Lisp accept only ternary **if**; Le-Lisp supports at least binary **if** with a **progn** implicit in the alternative. (**progn** corresponds to Scheme **begin**.)

Quotation consists of returning, as a value, that term following the keyword. That practice is clearly articulated in this fragment of code:

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

You might well ask whether there is a difference between implicit and explicit quotation, for example, between `33` and `'33` or even between `#(fa do sol)` and `'#(fa do sol)`.[7] The first comparison—between `33` and `'33`—impinges on immediate objects although the second one—between `#(fa do sol)` and `'#(fa do sol)`—impinges on composite objects (though they are atomic objects in Lisp terminology). It is possible to imagine divergent meanings for those two fragments. Explicit quotation simply returns its quotation as its value whereas `#(fa do sol)` could return a new instance of the vector for every evaluation—a new instance of a vector of three components initialized by three particular symbols. In other words, `#(fa do sol)` may be nothing other than the abbreviation of `(vector 'fa 'do 'sol)` (that's one of the possibilities in Scheme, though not the right one), and its behavior is quite different from `'#(fa do sol)` and from `(vector fa do sol)`, for that matter. We'll be returning later [see p. 140] to the question of what meaning to give to quotation, since, as you can see, the subject is far from simple.

1.4.2 Alternatives

As we look at *alternatives*, we'll consider the special form `if` as ternary, a control structure that evaluates its first argument (the *condition*), then according to the value it gets from that evaluation, chooses to return the value of its second argument (the *consequence*) or its third argument (the *alternate*). That idea is expressed in this fragment of code:

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

This program does not do full justice to the representation of Booleans. As you have no doubt noticed, we're mixing two languages here: the first is Scheme (or at least, a close enough approximation that it's indistinguishable from Scheme) whereas the second is also Scheme (or at least something quite close). The first language implements the second. As a consequence, there is the same relation between them as, for example, between Pascal (the language of the first implementation of TEX) and TEX itself [Knu84]. Consequently, there is no reason to identify the representation of Booleans of these two languages.

The function `evaluate` returns a value belonging to the language that is being defined. That value maintains no *a priori* relation to the Boolean values of the defining language. Since we follow the convention that any object different from the Boolean *false* must be considered as Boolean *true*, a more carefully written program would look like this:

```
... (case (car e) ...
```

7. In Scheme, the notation `#( ... )` represents a quoted vector.

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

Of course, we assume that the variable **the-false-value** has as its value the representation in the defining language of the Boolean *false* in the language being defined. There's a wide choice available to us for this value; for example,

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

The comparison of any value to the Boolean *false* is carried out by the physical comparer **eq?**, so a dotted pair handles the issue very well and can't be confused with any other possible value in the language being defined.

This discussion is not really trivial. In fact, Lisp chronicles are full of disputes about the differences between the Boolean value *false*, the empty list `()`, and the symbol **NIL**. The cleanest position to take in this controversy, quite independent of whether or not to preserve existing practice, is that *false* is different from `()`—which is, after all, simply an empty list—and that those two have nothing to do with the symbol spelled **N**, **I**, **L**.

That is, in fact, the position that Scheme takes; the position was adopted several weeks before being standardized by IEEE (the Institute of Electrical and Electronic Engineers) [IEE91].

Where things get worse is that `()` is pronounced *nil*! In "traditional" Lisp, *false*, `()`, and **NIL** are all one and same symbol. In Le-Lisp, **NIL** is a variable, its value is `()`, and the empty list has been assimilated with the Boolean *false* and with the empty symbol `||`.

1.4.3 Sequence

A *sequence* makes it possible to use a single syntactic form for a group of forms to evaluate sequentially. Like the well known **begin** ... **end** of the family of languages related to Algol, Scheme prefixes this special form by **begin** whereas other Lisps use **progn**, a generalization of **prog1**, **prog2**, etc. The sequential evaluation of a group of forms is "subcontracted" to a particular function: **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) )
      '() ) )
```

With that definition, the meaning of the sequence is now canonical. Nevertheless, we should note that in the middle of the definition of **eprogn**, there is a tail recursive call to **evaluate** to handle the final term of the sequence. That computation of the final term of a sequence is carried out as though it, and it alone, replaced the entire sequence. (We'll talk more about tail recursion later. [see p. 104])

We should also note the meaning we have to attribute to `(begin)`. Here, we've defined `(begin)` to return the empty list `()`, but why should we choose the empty list? Why not something else, anything else, like `bruce` or `brian`[8]? This choice is part of our heritage from Lisp, where the prevailing custom returns `nil` when nothing better seems to be obligatory. However, in a world where *false*, `()`, and `nil` are distinct, what should we return? We're going to specialize our language as it's defined in this chapter so that `(begin)` returns the value `empty-begin`, which will be the (almost) arbitrary number `813`[9] [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)
```

Our problem comes from the fact that the implementation that we are defining must necessarily return a value. Like Scheme, the language could attribute no particular meaning to `(begin)`; that choice could be interpreted in at least two different ways: either this way of writing is permitted within a particular implementation, in which case it is an extension that must return a value freely chosen by the implementation in question; or this way of writing is not allowed and thus an error. In light of the consequences, it's better to avoid using this form when no guarantee exists about its value. Some implementations have an object, `#<unspecified>`, that lends itself to this use, as well as more generally to any situation where we don't know what we should return because nothing seems appropriate. That object is usually printable; it should not be confused with the undefined pseudo-value. **[see p. 60]**

Sequences are of no particular interest if a language is purely functional (that is, if it has no side effects). What is the point of evaluating a program if we don't care about the results? Well, in fact there are situations in which we use a computation simply for its side effects. Consider, for example, a video game programmed in a purely functional language; computations take time, whether we are interested in their results or not; it may be just this very side effect—slowing things down—rather than the result that interests us. Provided that the compiler is not smart enough to notice and remove any useless computation, we can use such side effects to slow the program down sufficiently to accomodate a player's reflexes, say.

In the presence of conventional read or write operations, which have side effects on data flow, sequencing becomes very interesting because it is obviously clearer to pose a question (by means of `display`), then wait for the response (by means of `read`) than to do the reverse. Sequencing is, in that sense, the explicit form for putting a series of evaluations in order. Other special forms could also introduce a certain kind of order. For example, alternatives could do so, like this:

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

8. For the Monty Python fans in the audience
9. Fans of the gentleman thief Arsene Lupin will recognize the appropriateness of this choice.

That rule,[10] however, could also be simulated by this:

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

That last rule shows—in case you weren't convinced yet—that **begin** is not really a necessary special form in Scheme since it can be simulated by the functional application that forces arguments to be computed before the body of the invoked function (the *call by value* evaluation rule).

1.4.4 Assignment

As in many other languages, the value of a variable can be modified; we say then that we assign the variable. Since this assignment involves modifying the value in the environment of the variable, we leave this problem to the function **update!**.[11] We'll explain that function later, on page 129.

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

Assignment is carried out in two steps: first, the new value is calculated; then it becomes the value of the variable. Notice that the variable is not the result of a calculation. Many semantic variations exist for assignment, and we'll discuss them later. [see p. 11] For the moment, it's important to bear in mind that the value of an assignment is not specified in Scheme.

1.4.5 Abstraction

Functions (also known as *procedures* in the jargon of Scheme) are the result of evaluating the special form, **lambda**, a name that refers to λ-calculus and indicates an *abstraction*. We delegate the chore of actually creating the function to **make-function**, and we furnish all the available parameters that **make-function** might need, namely, the list of variables, the body of the function, and the current environment.

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

1.4.6 Functional Application

When a list has no special operator as its first term, it's known as a *functional application*, or, referring to λ-calculus again, a *combination*. The function we get by evaluating the first term is applied to its arguments, which we get by evaluating the following terms. The following code reflects that very idea.

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

10. The variable *void* must not be free in β. That condition is trivially satisfied if *void* does not appear in β. For that reason, we usually use **gensym** to create new variables that are certain to have not yet been used. See Exercise 1.11.

11. According to current practice in Scheme, functions with side effects have a name suffixed by an exclamation point.

The utility function **evlis** takes a list of expressions and returns the corresponding list of values of those expressions. It is defined like this:

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

The function **invoke** is in charge of applying its first argument (a function, unless an error occurs) to its second argument (the list of arguments of the function indicated by the first argument); it then returns the value of that application as its result. In order to clarify the various uses of the word "argument" in that sentence, notice that **invoke** is similar to the more conventional **apply**, outside the explicit eruption of the environment. (We'll return later in Section 1.6 **[see p. 15]** to the exact representation of functions and environments.)

More about evaluate

The explanation we just walked through is more or less precise. A few utility functions, such as **lookup** and **update!** (that handle environments) or **make-function** and **invoke** (that handle functions) have not yet been fully explained. Even so, we can already answer many questions about **evaluate**. For example, it is already apparent that the dialect we are defining has only one unique name-space; that it is mono-valued (like Lisp_1) **[see p. 31]**; and that there are functional objects present in the dialect. However, we don't yet know the order of evaluation of arguments.

The order in which arguments are evaluated in the Lisp that we are defining is similar to the order of evaluation of arguments to **cons** as it appears in **evlis**. We could, however, impose any order that we want (for example, left to right) by using an explicit sequence, like this:

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

Without enlarging the arsenal[12] that we're using in the defining language, we have increased the precision of the description of the Lisp we've defined. This first part of this book tends to define a certain dialect more and more precisely, all the while restricting more and more the dialect serving for the definition.

1.5 Representing the Environment

The environment associates values with variables. The conventional data structure in Lisp for representing such associations is the *association list*, also known as the *A-list*. We're going to represent the environment as an A-list associating values

12. As you know, `let` is a macro that expands into a functional application: `(let ((`x π_1`))`π_2`)` $\equiv$ `((lambda (`x`)` π_2`)` π_1`)`.

and variables. To simplify, we'll represent the variables by symbols of the same name.

In this way, we can define the functions **lookup** and **update!** very easily, like this:

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

We see a second[13] kind of error appear when we want to know the value of an unknown variable. Here again, we'll just use a call to **wrong** to express the problem that the interpreter confronts.

Back in the dark ages, when interpreters had very little memory[14] to work with, implementors often favored a generalized mode of *autoquote*. A variable without a value still had one, and that value was the symbol of the same name. It's disheartening to see things that we attached so much importance to separating—like symbol and variable—already getting back together and re-introducing so much confusion.

Even if it were practical for implementers never to provoke an error and thus to provide an idyllic world from which error had been banished, that design would still have a major drawback because the goal of a program is not so much to avoid committing errors but rather to fulfil its duty. In that sense, an error is a kind of crude guard rail: when we encounter it, it shows us that the program is not going as we intended. Errors and erroneous contexts need to be pointed out as early as possible so that their source can be corrected as soon as possible. The *autoquote* mode is consequently a poor design-choice because it lets certain situations worsen without our being able to see them in time to repair them.

The function **update!** modifies the environment and is thus likely to provoke the same error: the value of an unknown variable can't be modified. We'll come back to this point when we talk about the global environment.

```
(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) ) )
```

The contract that the function **update!** abides by foresees that the function returns a value which will become that final special form of assignment. The value of an assignment is not defined in Scheme. That fact means that a portable program cannot expect a precise value, so there are many possible return values. For example,

13. The first kind of error appeared on page 6.

14. Memory, along with input/output facilities, still remains the most expensive part of a computer, even though the price keeps falling.

1. the value that has just been assigned; (That's what we saw earlier with `update!`.)
2. the former contents of the variable; (This possibility poses a slight problem with respect to initialization when we give the first value to a variable.)
3. an object representing whatever has not been specified, with which we can do very little—something like `#<UFO>`;
4. the value of a form with a non-specified value, such as the form `set-cdr!` in Scheme.

The environment can be seen as a composite abstract type. We can then extract or modify subparts of the environment with selection and modification functions. We then still have to define how to construct and enrich an environment.

Let's start with an empty initial environment. We can represent that idea simply, like this:

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

(Later, in Section 1.6, we'll make the effort to produce a standard environment a little richer than that.)

When a function is applied, a new environment is built, and that new environment binds variables of that function to their values. The function **extend** extends an environment **env** with a list of **variables** and a list of **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 less values") ) )
        ((null? variables)
             (if (null? values)
                 env
                 (wrong "Too much values") ) )
        ((symbol? variables) (cons (cons variables values) env)) ) )
```

The main difficulty is that we have to analyze syntactically all the possible forms that a *<list-of-variables>* can have within an abstraction in Scheme.[15] A *list of variables* is represented by a list of symbols, possibly a dotted list, that is, terminated not by () but by a symbol, which we call a *dotted variable*. More formally, a list of variables corresponds to the following pseudo-grammar:

<list-of-variables>	:=	()
	|	*<variable>*
	|	(*<variable>* . *<list-of-variables>*)
<variable>	∈	**Symbol**

When we extend an environment, there must be agreement between the names and values. Usually, there must be as many values as there are variables, unless

15. Certain Lisp sytems, such as COMMON LISP, have succombed to the temptation to enrich the list of variables with various keywords, such as **&aux**, **&key**, **&rest**, and so forth. This practice greatly complicates the binding mechanism. Other systems generalize the binding of variables with pattern matching [SJ93].

the list of variables ends with a dotted variable or an *n-ary variable* that can take all the superfluous values in a list. Two errors can thus be raised, depending on whether there are too many or too few values.

1.6 Representing Functions

Perhaps the easiest way to represent functions is to use functions. Of course, the two instances of the word "function" in that sentence refer to different entities. More precisely, we should say, "The way to represent functions in the language that is being defined is to use functions of the defining language, that is, functions of the implementation language." The representation that we've chosen here minimizes the calling protocol in this way: the function **invoke** will have to verify only that its first argument really is a function, that is, an object that can be applied.

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

We really can't get any simpler than that. You might ask why we have specialized the function **invoke**, already a simple definition and merely called once from **evaluate**. The reason is that here we're setting the general structure of the interpreters that we're going to stuff you with later, and we will see other ways of coding that are simpler but more efficient and that require a more complicated form of **invoke**. You could also tackle Exercises 1.7 and 1.8 now.

A new kind of error appears here when we try to apply an object that is not a function. This kind of error can be detected at the moment of application, that is, after the evaluation of all the arguments. Other strategies would also be possible; for example, to attempt to warn the user as early as possible. In that case, we could impose an order in the evaluation of function applications, like this:

1. evaluate the term in the functional position;
2. if that value is not applicable, then raise an error;
3. evaluate the arguments from left to right;
4. compare the number of arguments to the arity of the function to apply and raise an error if they do not agree.

Evaluating arguments in order from left to right is useful for those who read left to right. It is also easy to implement since the order of evaluation is then obvious, but it complicates the task of the compiler. If the compiler tries to evaluate the arguments in a different order (for example, to improve register allocations), the compiler must prove that the new order of expressions does not change the semantics of the program.

Of course, we could attempt to do even better, by checking the arity sooner, like this:

1. evaluate the term in the functional position;
2. if that value is not applicable, then raise an error; otherwise, check its arity;

3. evaluate the arguments from left to right as long as their number still agrees with the arity; otherwise, raise an error;
4. apply the function to its arguments.[16]

COMMON LISP insists that arguments must be evaluated from left to right, but for reasons of efficiency, it allows the term in the function position to be evaluated either before or after the others.

Scheme, in contrast, does not impose an order of evaluation for the terms of a functional application, and Scheme does not distinguish the function position in any particular way. Since there is no imposed order, the evaluator is free to choose whatever order it prefers; the compiler can then re-order terms without worrying. **[see p. 164]** The user no longer knows which order will be chosen, and so must use **begin** when certain effects must be kept in sequence.

As a matter of style, it's not very elegant to use a functional application to put side effects in sequence. For that reason, you should avoid writing such things as `(f (set! f π) (set! f π'))` where the function that will actually be applied can't be seen. Errors arising from that kind of practice, that is, errors due to the fact that the order of evaluation cannot be known, are extremely hard to detect.

The Execution Environment of the Body of a Function

Applying a function comes down to evaluating its body in an environment where its variables are bound to values that they have assumed during the application. Remember that during the call to **make-function**, we provided it with the parameters that were made available by **evaluate**. Throughout the rest of this section, we'll highlight the various environments that come into play by mentioning them in *italic*.

Minimal Environment

Let's first try a definition of a stripped down, minimal environment.

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

In conformity with the contract that we explained earlier, the body of the function is evaluated in an environment where the variables are bound to their values. For example, the combinator K defined as `(lambda (a b) a)` can be applied like this:

`(K 1 2)` → `1`

The nuisance here is that the means available to a function are rather minimal. The body of a function can utilize nothing other than its own variables since the initial environment, *env.init*, is empty. It does not even have access to the global environment where the usual functions, such as **car**, **cons**, and so forth, are defined.

16. The function could then examine the types of its arguments, but that task does not have to do with the function call protocol.

Patched Environment

Let's try again with an enriched environment patched like this:

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

This new version lets the body of the function use the global environment and all its usual functions. Now how do we globally define two functions that are mutually recursive? Also, what is the value of the expression on the left (macroexpanded on the right)?

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

Let's look in detail at how that expression is evaluated:

```
((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
```

The body of the internal function `(lambda (b) (list a b))` is evaluated in an environment that we get by extending the global environment with the variable `b`. That environment lacks the variable `a` because it is not visible, so this try fails, too!

Improving the Patch

Since we need to see the variable `a` within the internal function, it suffices to provide the current environment to **`invoke`**, which in turn will transmit that environment to the functions that are called. In consequence, we now have an idea of a current environment kept up to date by **`evaluate`** and **`invoke`**, so let's modify these functions. However, let's modify them in such a way that we don't confuse these new definitions with the previous ones; we'll use a prefix, `d.` to avoid confusion, like this:

```
(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)) ) )
```

Here we notice that in the definition of `d.invoke`, it is no longer really useful to provide the definition environment **env** to the *def.env* variable since only the current environment, *current.env*, is being used.

Now if we look at the same expression, highlighted with the name of the environment being used, we see that our example works like this:

```
((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
```

Of course, in this example, we clearly see the stack discipline that the bindings are following. Every binding form pushes its new bindings onto the current environment and pops them off after execution.

Fixing the Problem

There is still a problem, though. To see it, look at this variation:

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

The function `(lambda (b) (list a b))` is created in an environment where `a` is bound to `1`, but at the time the function is applied, the current environment is extended by the sole binding to `b`, and once again, `a` is absent from that environment. As a consequence, the function `(lambda (b) (list a b))` once again forgets or loses its variable `a`.

No doubt you noticed in the previous definition of `d.make-function` that two environments were present: the definition environment for the function, *def.env*, and the calling environment of the function, *current.env*. Two moments are important in the life of a function: its creation and its application(s). Granted, there is only one moment when the function is created, but there can very well be many times when the function is applied; or indeed, there may be no moment when it is applied! As a consequence, the only[17] environment that we can associate with a function with any certainty is the environment when it was created. Let's go back

17. True, we could leave it up to the program to choose explicitly which environment the function should end up in. See the form `closure` on page 113.

to the original definitions of the functions **evaluate** and **invoke**, and this time, let's write **make-function** this way:

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

All the examples now behave well, and in particular, the preceding example now has the following evaluation trace:

```
(((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
```

The form `(lambda (b) (list a b))` is created in the global environment extended by the variable `a`. When the function is applied in the global environment, it extends its own environment of definition by `b`, and thus permits the body of the function to be evaluated in an environment where both `a` and `b` are known. When the function returns its result, the evaluation continues in the global environment. We often refer to the value of an abstraction as its *closure* because the value closes its definition environment.

Notice that the present definition of **make-function** itself uses closure within the definition language. That use is not obligatory, as we'll see later in Chapter 3. [see p. 92] The function **make-function** has a closure for its value, and that fact is a distinctive trait of higher-order functional languages.

1.6.1 Dynamic and Lexical Binding

There are at least two important points in that discussion about environments. First, it demonstrates clearly how complicated the issues about environments are. Any evaluation is always carried out within a certain environment, and the management of the environment is a major point that evaluators must resolve efficiently. In Chapter 3, we will see more complicated structures, such as escapes and the form **unwind-protect**, that oblige us to define very precisely which environments are under consideration.

The second point concerns the last two variations in the previous section which are characteristic of *dynamic*[18] and *lexical* binding. In a *lexical* Lisp, a function evaluates its body in its own definition environment extended by its variables, whereas in a *dynamic* Lisp, a function extends the current environment, that is, the environment of the application.

18. In the context of object-oriented languages, the term *dynamic binding* usually refers to the fact that the method associated with a message is determined by the dynamic type of the object to which the message is sent, rather than by its static type.

Current fashion favors lexical Lisps, but you mustn't conclude from that fact that dynamic languages have no future. For one thing, very useful dynamic languages are still widely in use, such languages as TEX [Knu84], GNU EMACS LISP [LLSt93], or Perl [WS90].

For another, the idea of dynamic binding is an important concept in programming. It corresponds to establishing a valid binding when beginning a computation and that binding is undone automatically in a guaranteed way as soon as that computation is complete.

This programming strategy can be employed effectively in forward-looking computations, such as, for example, those in artificial intelligence. In those situations, we pose a hypothesis, and we develop consequences from it. When we discover an incoherence or inconsistency, we must abandon that hypothesis in order to explore another; this technique is known as backtracking. If the consequences have been carried out with no side-effects, for example in such structures as A-lists, then abandoning the hypothesis will automatically recycle the consequences, but if, in contrast, we had used physical modifications such as global assignments of variables, modifications of arrays, and so forth, then abandoning a hypothesis would entail restoring the entire environment where the hypothesis was first formulated. One oversight in such a situation would be fatal! Dynamic binding makes it possible to insure that a dynamic variable is present and correctly assigned during a computation and only during that computation regardless of the outcome. This property is heavily used for exception handling.

Variables are programming entities that have their own *scope*. The scope of a variable is essentially a geographic idea corresponding to the region in the programming text where the variable is visible and thus is accessible. In pure Scheme (that is, unburdened with superfluous but useful syntax, such as `let`), only one binding form exists: `lambda`. It is the only form that introduces variables and confers on them a scope limited strictly to the body of the function. In contrast, the scope of a variable in a dynamic Lisp has no such limitation *a priori*. Consider this, for example:

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

In a lexical Lisp, the variable `y` in `foo`[19] is a reference to the global variable `y`, which in no way can be confused with the variable `y` in `bar`. In dynamic Lisp, the variable `y` in `bar` is visible (indeed, it is seen) from the body of the function `foo` because when `foo` is invoked, the current environment contains that variable `y`. Consequently, if we give the value `0` to the global variable `y`, we get these results:

```
(define y 0)
(list (bar 100) (foo 3))    → ((1991 0) (3 0))      in lexical Lisp
(list (bar 100) (foo 3))    → ((1991 100) (3 0))    in dynamic Lisp
```

Notice that in dynamic Lisp, `bar` has no means of knowing that the function `foo` that `bar` calls references its own variable `y`. Conversely, the function `foo` doesn't know where to find the variable `y` that it references. For this reason, `bar` must put `y` into the current environment so that `foo` can find it there. Just before `bar` returns, `y` has to be removed from the current environment.

19. For the etymology of foo, see [Ray91].

Of course, in the absence of free variables, there is no noticeable difference between dynamically scoped and lexically scoped Lisp.

Lexical binding is known by that name because we can always start simply from the text of the function and, for any variable, either find the form that bound the variable or know with certainty that it is a global variable. The method is so simple that we can point with a pencil (or a mouse) at the variable and go right along from right to left, from bottom to top, until we find the first binding form around this variable. The name dynamic binding plays on another concept, that of the *dynamic extent*, which we'll get to later. [see p. 77]

Scheme supports only lexical variables. COMMON LISP supports both kinds of binding with the same syntax. EULISP and IS-Lisp clearly distinguish the two kinds syntactically in two separate name spaces. [see p. 43]

Scope may be obscured locally by *shadowing*. Shadowing occurs when one variable hides another because they both have the same name. Lexical binding forms are nested or disjoint from one another. This well known "block" discipline is inherited from Algol 60.

Under the inspiration of λ-calculus, which loaned its name to the special form `lambda` [Per79], Lisp 1.0 was defined as a dynamic Lisp, but early on, John McCarthy recognized that he expected the following expression to return `(2 3)` rather than `(1 3)`.

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

That anomaly (dare we call it a bug?) was corrected by introducing a new special form, known as `function`. Its argument was a `lambda` form, and it created a *closure*, that is, a function associated with its definition environment. When that closure was applied, instead of extending the current environment, it extended its own definition environment that it had closed (in the sense of preserved) within itself. In programming terms, the special form `function`[20] is defined accordingly with `d.evaluate` and `d.invoke`, like this:

```
(define (d.evaluate e env)
  (if (atom? e) ...
      (case (car e)
        ...
        ((function)   ;Syntax: (function (lambda variables body))
         (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 )) ) ) )
```

20. This simulation is not exactly correct in the sense that there are many dialects (such as CLtL1 in [Ste84]) where `lambda` is not a special operator, but a keyword, a kind of syntactic marker similar to `else` that can appear in Scheme in a `cond` or a `case`. `lambda` does not require `d.evaluate` for it to be handled correctly. `lambda` forms may also be restricted to appear exclusively in the first term of functional applications, accompanied by `function`, or in function definitions.

```
(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) ) )
```

That's not the end of the story, however. That `function` was regarded as a convenience that the end-user had to rely on because of an inadequate implementation. The early compilers very quickly caught on to the fact that, in terms of performance, lexical environments have a great advantage—as one might expect in compilation—and that in any case, wherever the variables were during execution, the compiler could generate a more or less direct access rather than searching dynamically for the value. By default, then, all the variables of compiled functions were treated as lexical, except those that were explicitly declared as dynamic, or in the jargon of the time, *special*. The declaration `(declare (special x))` solely for the use of the compiler in Lisp 1.5, MacLisp, COMMON LISP, and so forth, designated the variable x as having special behavior.

Efficiency was not the only reason for this turn of events. There was also a loss of *referential transparency*. Referential transparency is the property that a language has when substituting an expression in a program for an equivalent expression does not change the behavior of the program; that is, they both calculate the same thing; either they return the same value, or neither of the two terminates. For example, consider this:

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

Referential transparency is lost once a language has side effects. In such a case, we need new, more precise definitions for equivalence as a relation in order to talk about referential transparency. Scheme—without assignment, without side effects, without continuations—is referentially transparent. [see **Ex. 3.10**] This property is also a goal that we work toward when we are trying to write programs that are genuinely re-usable, in the sense of depending as little as possible on the context where they are used.

The variables of a function such as `(lambda (u) (+ u u))` are what we conventionally call *silent*. Their names have no particular importance and can be replaced by any other name. The function `(lambda (n347) (+ n347 n347))` is nothing more nor less[21] than the function `(lambda (u) (+ u u))`.

We're still waiting for the language that respects this invariant. There's nothing like that in dynamic Lisp. Just consider this:

```
(define (map fn l)          ; mapcar in Lisp
  (if (pair? l)
      (cons (fn (car l)) (map fn (cdr l)))
```

21. In the technical terms of λ-calculus, this change of names for variables is known as an α-conversion.

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

(The function `list-ref` extracts the element at index n from a list.)

In Scheme, the result would be `(c b a)`, but in a dynamic Lisp, the result would be `(0 0 0)`. The reason: the variable `l` is free in the body of `(lambda (x) (list-ref l x))` but that variable is captured by the variable `l` in `map`.

We could resolve that problem simply by changing the names that are in conflict. For example, it suffices to rename one of those two variables, `l`. We might perhaps choose to rename the variable in `map` since it is probably the more useful, but what name could we choose that would guarantee that the problem would not crop up again? If we prefix the names of variables by the social security number of the programmer and suffix them by the standardized time (indicated, say, by the number of hundredths of seconds elapsed since noon, 1 January 1901), we obviously lower the risk of name-collisions, but we lose something in readability in our programs.

The situation at the beginning of the eighties was particularly sensitive. We taught Lisp to students by observing its interpreter, which differed from its compiler in this fundamental point about the value of a variable. From 1975 on, Scheme [SS75] had shown that we could reconcile an interpreter and compiler and then live in a completely lexical world. COMMON LISP buried the problem by postulating that *good* semantics were compiler semantics, so everything should be lexical by default. An interpreter just had to conform to these new canonical laws. The increasing success of Scheme, of functional languages such as ML, and of their spin-offs first spread and then imposed this new view.

1.6.2 Deep or Shallow Implementation

Things are not so simple, however, and implementers have found ways of increasing how fast the value of a dynamic variable is determined. When the environment is represented as an association-list, the cost of searching[22] for the value of a variable (that is, the cost of the function `lookup`) is linear with respect to the length of the list. This mechanism is known as *deep binding*.

Another technique, known as *shallow binding*, also exists. Shallow binding occurs when each variable is associated with a place where its value is always stored independently of the current environment. The simplest implementation of this idea is that the location should be a field in the symbol associated with the variable; it's known as `Cval`, or the *value cell*. The cost of `lookup` is constant in that case since the cost is based on an indirection, possibly followed by an additional offset. Since there's rarely a gain without a loss, we have to admit here that a function call is more costly in this model since it has to save the values of variables that are going to be bound and modify the symbols associated with those variables

22. Fortunately, statistics show that we search more often for the first variables than for those that are buried more deeply. Besides, it's worth noting that lexical environments are smaller than dynamic environments, since dynamic environments have to carry around all the bindings that are being computed [Bak92a].

so that the symbols contain the new values. When the function returns, moreover, it must restore the old values in the symbols that they came from—a practice that can compromise tail recursion. (But see [SJ93] for alternatives.)

We can partially simulate[23] shallow binding by changing the representation of the environment. In what follows, we'll assume that lists of variables are not dotted. This assumption will make it easier to decode lists of variables. We'll also assume that we don't need to verify the arity of functions. These new functions will be prefixed by `s.` to make them easier to recognize.

```
(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) )
```

In Scheme, the functions **putprop** and **getprop** are not standard because they cause highly inefficient global side effects, but they resemble the functions **put** and **get** in [AS85]. [see **Ex. 2.6**]

Here, the functions **putprop** and **getprop** simulate that field[24] where a symbol stores the value of a variable of the same name. Independently[25] of their actual implementation, these functions should be regarded as though they have constant cost.

Notice that in the preceding simulation, the environment **env** has completely disappeared because it no longer serves any purpose. This disappearance means that we have to modify the implementation of closures since they can no longer close the environment (since it doesn't exist any longer). The technique for doing so (which we'll see later) consists of analyzing the body of the function to identify free variables and then treating them in the appropriate way.

Deep binding favors changing the environment and programming by multitasking, to the detriment of searching for values of variables. Shallow binding favors searching for the values of variables to the detriment of function calls. Henry Baker

23. However, we are not treating the assignment of a closed variable here. For that topic, see [BCSJ86].

24. The name of the property being used, **apval** [see p. 31], was chosen in honor of the name used in [MAE+62] when these values really were stored in P-lists.

25. The functions sweep down a symbol's list of properties (its P-list) until they find the right property. The cost of this search is linear with respect to the length of the P-list, unless the implementation associates symbols with properties by means of a hash table.

[Bak78] combined the two in the technique of *rerooting*.

Remember, finally, that deep binding and shallow binding are merely implementation techniques that have nothing to do with the semantics of binding.

1.7 Global Environment

An empty global environment is a poor thing, so most Lisp systems supply *libraries* to fill it up. There are, for example, more than 700 functions in the global environment of COMMON LISP (CLtL1); more than 1,500 in Le-Lisp; more than 10,000 in ZetaLisp, etc. Without its library, Lisp would be only a kind of λ-calculus in which we couldn't even print the results of calculations. The idea of a library is important for the end-user. Whereas the special forms are the essence of the language from the point of view of any one producing the evaluator, it's the libraries that make a real difference for the end-user. Lisp folk history insists that it was the lack of such banalities as a library of trigonometric functions that made number crunchers drop Lisp early on. Their feeling, according to [Sla61], was that it might be a good thing to know how to integrate or differentiate symbolically but without sine or tangent, what could anyone really do with the language?

We expect to find all the usual functions, such as `car`, `cons`, etc., in the global environment. We may also find simple variables whose values are well known data, such as Boolean values and the empty list.

Now we are going to define two macros—just for convenience at this point, since we haven't even talked about macros yet,[26] important as they are. In fact, macros are such a significant and complicated phenomenon that we will devote an entire chapter to them later. **[see p. 311]**

The two macros that we'll define here will make it easier to elaborate the global environment. We'll define the global environment as an extension of the empty initial environment, `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)        ; The real apply of Scheme
              (wrong "Incorrect arity"
```

26. It is not our intention to put the whole book in its first chapter.

```
(list 'name values) ) ) ) ) ) )
```

Now we'll define a few very useful constants, though none of these three appears in standard Scheme. We note here that `t` is a variable in the Lisp that is being defined, while `#t` is a value in the Lisp that we are using as the definition language. Any value other than that of `the-false-value` is true.

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

Though it's useful to have a few global variables that let us get the real objects that represent Booleans or the empty list, another solution is to develop a syntax appropriate for doing that. Scheme uses the syntax `#t` and `#f`, and the values of those are the Booleans `true` and `false`. The point of those two is that they are always visible and that they cannot be corrupted.

1. They are always visible because we can write `#t` for *true* in any context, even if a local variable is named `t`.
2. They are incorruptible, an important fact, since many evaluators authorize alterations in the value of the variable `t`.

Such alterations lead to puzzles like this one: `(if t 1 2)` will have the value 2 in `(let ((t #f)) (if t 1 2))`.

Many solutions to that problem are possible. The simplest is to lock the immutability of these constants into the evaluator, like this:

```
(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" e)) )
      ... ) )
```

We could also introduce the idea of *mutable* and *immutable* binding. An immutable binding could not be the object of an assignment. Nothing could ever change the value of a variable that has been immutably bound. That concept exists, even if it is somewhat obscured, in many systems. The idea of *inline* functions designates those functions (also known as *integrable* or *open coded*) [see p. 199] where the body, appropriately instantiated, can replace the call.

To replace `(car x)` by the code that extracts the contents of the `car` field from the value of the dotted pair `x` implies the important hypothesis that nothing ever alters, has never altered, will never alter the value of the global variable `car`. Imagine the misfortune that the following fragment of a session illustrates.

```
(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) )
→  ?????
```

Fortunately again, the response can only be `a` or `b`. If `my-test` uses the value of `car` current at the time `my-test` was defined, the response would be `a`. If `my-test` uses the current value of `car`, the response would be `b`. It's helpful to compare the behavior of `my-test` and `my-global`, knowing that we'll usually see the first kind of behavior for `my-test` when the evaluator is a compiler, and usually the second kind of behavior for `my-global`. [see p. 54]

We'll also add a few working variables[27] to the global environment that we've been building because there is no dynamic creation of global variables in the present evaluator. The names that we suggest here cover roughly 96.037% of the names of functions that Lispers testing a new evaluator usually come up with spontaneously.

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

Finally, we'll define a few functions, but not all of them, because listing all of them would put you to sleep. Our main difficulty now is to adapt the primitives of Lisp to the calling protocol of the Lisp that is being defined. Knowing that the arguments are all brought together into one list by the interpreter, we simply have to use `apply`.[28] Note that the arity of the primitive will be respected because of the way `defprimitive` expands it.

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

1.8 Starting the Interpreter

The only thing left to tell you is how to get into this new world that we've defined.

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

Since our interpreter is still open to innovation, we suggest an exercise in which you implement a function for exiting.

27. These variables are, unfortunately, initialized here. This fault will be corrected later.
28. Once again, we congratulate ourselves that we did not call `invoke` "`apply`."

1.9 Conclusions

Have we really defined a language at this point?

No one could doubt that the function **evaluate** can be started, that we can submit expressions to it, and that it will return their values, once its computations are complete. However, the function **evaluate** itself makes no sense apart from its definition language, and, in the absence of a definition of the definition language, nothing is sure. Since every true Lisper has an in-born reflex for bootstrapping, it's probably sufficient to identify the definition language as the language that we've defined. In consequence, we now have a language L defined by a function **evaluate**, written in the language L. The language defined that way is thus a solution to this equation in L:

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

For any program π, the evaluation of π in L (denoted $L\pi$) must behave the same way (and thus have the same value if π terminates) as the evaluation of the expression `(evaluate (quote` π`) env.global)`, all still in L. One amusing consequence of this equation is that the function **evaluate**[29] is capable of auto-interpretation. The following expressions are thus equivalent:

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

Are there any solutions to that equation? The answer is yes; in fact, there are many solutions. As we saw earlier, the order of evaluation is not necessarily apparent from the definition of **evaluate**, and many other properties of the definition language are unconsciously *inherited* by the language being defined. We can say next to nothing about them, in fact, because there are also a great many trivial solutions to the equation. Take, for example, the language L_{2001}; its semantics is that every program written in it has, as its value, the number 2001. That language trivially satisfies our equation. We have to depend on other methods, then, if we want to define a real language, and those other methods will be the subject of later chapters.

1.10 Exercises

Exercise 1.1 : Modify the function **evaluate** so that it becomes a tracer. All function calls should display their arguments and their results. You can well imagine extending such a rudimentary tracer to make a step-by-step debugger that could modify the execution path of the program under its control.

Exercise 1.2 : When **evlis** evaluates a list containing only one expression, it has to carry out a useless recursion. Find a way to eliminate that recursion.

29. It is necessary to clean all syntactic abbreviations and macros out of **evaluate**—such things as **define**, **case**, and so forth. Then we still have to provide a global environment containing the variables **evaluate**, **evlis**, etc.

Exercise 1.3 : Suppose we now define the function **extend** like this:

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

Define the associated functions, **lookup** and **update!**. Compare them with their earlier definitions.

Exercise 1.4 : Another way of implementing shallow binding was suggested by the idea of a *rack* in [SS80]. Instead of each symbol being associated with a field to contain the value of the variable of the same name, there is a stack for that purpose. At any given time, the value of the variable is the value found on top of that stack of associated values. Rewrite the functions **s.make-function**, **s.lookup**, and **s.update!** to take advantage of this new representation.

Exercise 1.5 : The definition of the primitive < is false! In practice, it returns a Boolean value of the implementation language instead of a Boolean value of the language being defined. Correct this fault.

Exercise 1.6 : Define the function **list**.

Exercise 1.7 : For those who are fond of continuations, define **call/cc**.

Exercise 1.8 : Define **apply**.

Exercise 1.9 : Define a function **end** so that you can exit cleanly from the interpreter we developed in this chapter.

Exercise 1.10 : Compare the speed of Scheme and **evaluate**. Then compare the speed of **evaluate** and **evaluate** interpreted by **evaluate**.

Exercise 1.11 : The sequence **begin** was defined by means of **lambda** [see p. 9] but it used **gensym** to avoid any possible captures. Redefine **begin** in the same spirit but do not use **gensym** to do so.

Recommended Reading

All the references to interpreters mentioned at the beginning of this chapter make interesting reading, but if you can read only a little, the most rewarding are probably these:

- among the λ-papers, [SS78a];
- the shortest article ever written that still presents an evaluator, [McC78b];

- to get a taste of non-pedantic formalism, [Rey72];
- to get to know the origins and beginnings, [MAE+62].

2
Lisp, 1, 2, ... ω

SINCE functions occupy a central place in Lisp, and because their efficiency is so crucial, there have been many experiments and a great deal of research about functions. Indeed, some of those experiments continue today. This chapter explains various ways of thinking about functions and functional applications. It will carry us up to what we'll call Lisp_1 or Lisp_2, their differences depending on the concept of separate name spaces. The chapter closes with a look at recursion and its implementation in these various contexts.

Among all the objects that an evaluator can handle, a function represents a very special case. This basic type has a special creator, **lambda**, and at least one legal operation: application. We could hardly constrain a type less without stripping away all its utility. Incidentally, this fact—that it has few qualities—makes a function particularly attractive for specifications or encapsulations because it is opaque and thus allows only what it is programmed for. We can, for example, use functions to represent objects that have fields and methods (that is, data members and member functions) as in [AR88]. Scheme-users are particularly appreciative of functions.

Attempts to increase the efficiency of functions have motivated many, often incompatible, variations. Historically, Lisp 1.5 [MAE+62] did not recognize the idea of a functional object. Its internals were such, by the way, that a variable, a function, a macro—all three—could co-exist with the same name, and at the same time, the three were represented with different properties (**APVAL**, **EXPR**, or **MACRO**[1]) on the P-list of the associated symbol.

MacLisp privileged named functions, and only recently, its descendant, COMMON LISP(CLtL2) [Ste90] introduced first class functional objects. In COMMON LISP, **lambda** is a syntactic keyword declaring something like, "Warning: we are defining an anonymous function." **lambda** forms have no value and can appear only in syntactically special places: in the first term of an application or as the first parameter of the special form **function**.

In contrast, Scheme, since its inception in 1975, has spread the ideas of a functional object and of a unique space of values by conferring the status of *first class*

1. **APVAL**, *A Permanent VALue*, stores the global value of a variable; **EXPR**, *an EXPRession*, stores a global definition of a function; **MACRO** stores a macro.

on practically everything. A first class object can be an argument or the value of a function; it can be stored in a variable, a list, an array, etc. The philosophy of Scheme is compatible with functional languages in the same class as ML, and we'll adopt the same philosophy here.

2.1 Lisp_1

The main activity in the previous chapter was to conform to that philosophy: the idea of a functional object prevailed there (**make-function** creates functional objects); and the process of evaluating terms in an application did not distinguish the function from its arguments; that is, the expression in the *function position* was not treated differently from the expressions in the following positions, those in the *parametric* positions. Let's look again at the most interesting fragments of that preceding interpreter.

```
(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) )) ) ) )
```

The salient points in it are these:

1. **lambda** is a special form creating first class objects; closures capture their definition environment.
2. All the terms in an application are evaluated by the same evaluator, namely, **evaluate**; **evlis** is simply **evaluate** mapped over a list of expressions.

That second characteristic makes Scheme into Lisp_1.

2.2 Lisp_2

Programs in Lisp are generally such that most functional applications have the name of a global function in their function position. That's certainly the case of all the programs in the preceding chapter. We could restrict the grammar of Lisp to impose a symbol in the **car** of every form. Doing so would not noticeably alter the look of the language, but it would imply that the evaluation that occurs for the term in the function position no longer needs all the complexity of **evaluate** and could get along with a mini-evaluator for those expressions—a mini-evaluator that knows how to handle only names of functions. Implementing this idea entails modifying the last clause in the preceding interpreter, like this:

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

We now have two different evaluators, one for each of the two positions where a variable may occur (that is, in functional or parametric position). Many different

kinds of behavior can correspond to one identifier, depending on its position; in this case, its position could be the position of a function or a parameter. If we specialize the function position, that specialization may be accompanied by the presence of a supplementary environment uniquely dedicated to functions. As a result, *a priori* it will be easier to look for the function associated with a name since this dedicated environment won't contain normal variables. The basic interpreter can then be rewritten to take into account these details. A function environment, **fenv**, and an evaluator specific to forms, **evaluate-application**, are clearly identified. We'll have two environments and two evaluators then, and so we'll adopt the name $Lisp_2$ [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 )) ) ) )
```

The evaluator dedicated to forms is **evaluate-application**; it receives the unevaluated function term, the evaluated arguments, and the two current environments. Notice that creating the function (by means of **lambda**) closes the two environments, both **end** and **fenv**, in a way that sets the values of free variables that occur in the body of functions, whether they appear in the position of a function or parameter. In other respects, this new version differs from the preceding one only by the fact that **fenv** accompanies **env**, the environment for variables, like a shadow. The functions **eprogn** and **evlis**, of course, have to be updated to propagate **fenv**, so they become this:

```
(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 ) )
```

When those functions are invoked, they bind their variables in the environment for variables, but only the way functions are created has been modified; the way they are invoked (by means of **invoke**) has not changed.

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

The task of the function evaluator is to analyze the function term in order to prepare the final invocation. If we keep the grammar of COMMON LISP, then only a symbol or a **lambda** form can appear in the function position.

```
(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)) ) )
```

What have we gained and lost by doing this? The first advantage is that searching for a function associated with a name is much easier than before because the search requires only a simple call to **lookup** now, and we thus eliminate a call to **f.evaluate** followed by determining syntactically that this is a reference. Moreover, since the function environment has been freed from all variables, it is certainly more compact, so searches can proceed more quickly there. A second advantage is that handling forms where there is a **lambda** form in the function position is greatly improved. Consider this example:

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

In that example, you can see that the closure corresponding to **(lambda (x) (* state-tax x))** will not be created; its body will be evaluated directly in the right environment.

The problem, though, is that the two advantages are not real gains since a simple analysis could produce the same effects in Lisp_1. The only real difference is in the implementation. Lisp_2 is a little bit more efficient because in it we can be sure that any name present in **fenv** is bound to a function. Each time that a binding enriches the function environment, we simply have to verify that the value present in the binding is really a function. Then we never again have to verify during use that it is really a function. Since every name has to belong to the initial function environment, we will bind it to a function calling **wrong** if it is invoked by accident.

Moreover, since every name is bound in **fenv** to a function, we can simplify the call to **invoke** to avoid the test in **(procedure? fn)**. We do that by keeping in mind that our implementation language is really Scheme (since what we are about to do is not legal in COMMON LISP because of the presence of a function

calculated in the form `((lookup fn fenv) args)`). We will define the function `evaluate-application` more precisely, like this:

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

In Lisp, the number of function calls is such that any sort of improvement in function calls is always for the good and can greatly influence overall performance. However, this particular gain is not particularly great since it comes into play only on *calculated* functional applications, that is, those that are not known statically, of which there are very few.

In contrast, on the down side, we have just lost the possibility of calculating the function to apply. The expression `(if` *condition* `(+ 3 4) (* 3 4))` can be factored in Scheme because of its common arguments `3` and `4`, and thus we can rewrite the expression as `((if` *condition* `+ *) 3 4)`. The rule for doing that is simple and highly algebraic. In fact, it is practically an identity, but in $Lisp_2$, that program is not even legal since the first term is neither a symbol nor a `lambda` form.

2.2.1 Evaluating a Function Term

The function environment introduced thus far is a long way from offering us the facilities of the environment for variables (the parametric environment). Particularly, as you saw in the preceding example, it does not let us calculate the function to apply. The traditional trick, prevailing at least as far back as MacLisp, was to enrich the function evaluator in order to send off any expressions that it did not understand to `f.evaluate`, so we have this:

```
(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                    ;** Modified **
               (f.evaluate fn env fenv) args env fenv )) ) )
```

Now we can solve our problem and write this:

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

That transformation is far from elegant since we have to add those disgraceful quotation marks to it, but at least it works. Yes, it works, but perhaps it works too well since the evaluator can now get into a loop.

`('''''''''1789` *arguments*`)`

The expression `'''''''''1789` is evaluated as many times as there are quotation marks; then it loops on the number 1789, with the number always equal to itself, never to a function. In short, the subcontracting we get from `f.evaluate` is a bit too well done and demands a bit more control. A variation on this problem exists as well when `evaluate-application` looks like this:

```
(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)) ) )
     ... ) )
```

In that variation, not all the symbols have been predefined in the initial function environment, **fenv.global**, and when a symbol has not been defined in the function environment, we search for its value in the environment for variables. Oops! Even if we assume that no such function **foo** exists, then we still get programs that can loop on the value of a variable, like this:

```
(let ((foo 'foo))
   (foo arguments) )
```

And it's a good idea to add a feature to **evaluate-application** to detect variables that have, as their value, the symbol that bears their name. Then we can again trick the function evaluator into looping even more viciously on lines like these:

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

The only clean solution is that first definition that we gave for the function evaluator, **evaluate-application** [see p. 34], and we have to search for a new method that will accept the calculation of the function term.

2.2.2 Duality of the Two Worlds

To summarize these problems, we should say that there are calculations belonging to the parametric world that we want to carry out in the function world, and vice versa. More precisely, we may want to pass a function as an argument or as a result, or we may even want the function that will be applied to be the result of a lengthy calculation.

If it is necessary to indicate a function in the function term, and if we want to calculate the function to apply, then it is sufficient to have a predefined function that knows how to apply functions, so let's introduce the function **funcall** (that is, *function call*). It applies its first argument (which ought to be a function) to its other arguments. Let's try to write our first program using **funcall**, like this:

`(if condition (+ 3 4) (* 3 4))` $\equiv$ `(funcall (if condition + *) 3 4)` *WRONG*

The arguments, especially the first one, are evaluated by the normal evaluator **f.evaluate**. The function **funcall** takes everything and carries out the application. We could easily define **funcall** like this:

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

In $Lisp_2$, the function **funcall** represents the calculated call. In all other cases, the function is known, and the verification of the fact that it is a function is no longer necessary.

In the definition of **funcall**, notice the call to **invoke**. It carries out that verification, in contrast to **evaluate-application**, where the verification has been suppressed. The function **funcall** resembles **apply** somewhat. Both take a function as the first argument and other arguments follow. The difference between them is that in a **funcall** form, we statically know the number of arguments that will be provided to the final function that is being invoked.

Unfortunately, there is still one more problem. When we write (**if** *condition* **+ ***), we want to get the addition or multiplication function as the result. But what we get right now is the value of the variable **+** or *****! In COMMON LISP these variables have nothing to do with any arithmetic operations whatsoever, but they are bound by the interaction mechanism to the last expression read and to the last result returned by the basic interaction loop (that is, *toplevel*)!

We introduced **funcall** because we wanted normal evaluation to lead to a result before behaving like a function. The reverse exists (which is really what we would like to have available) in a normal interpreter, of a result coming from the function evaluator. We want the value of the function variable **+**, such as **evaluate-application** would get, so we will introduce once again a new linguistic device: **function**. As a special form, **function** takes the name of a function and returns its functional value. In that way, we can jump between the two spaces and safely write the following:

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

To define **function**, we will add a supplementary clause to our interpreter, **f.evaluate**. The definition of **function** that follows has nothing to do with the similarly named one [see **p. 21**] that defined the syntax (**function** (**lambda** *variable body*)) to mark the creation of a closure. Here, we'll define (**function** *name-of-function*) to convert the name of a function into a functional value.

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

The definition of **function** could be extended, as it is in COMMON LISP, to handle forms like (**function** (**lambda** ...)), similar to what we say on page 21, but that is superfluous in the language that we've just defined because we have **lambda** available directly to do that. In COMMON LISP, that tactic is indispensable because there **lambda** is not really a special form, but rather a marker or syntactic keyword announcing that the definition of a function will follow. A special form prefixed by **lambda** can appear only in the function position or as an argument of the special form **function**.

The function **funcall** lets us take the result of a calculation coming from the parametric world and put it into the function world as a value. Conversely, the special form **function** lets us get the value of a variable from the function world.

There is a striking parallel between the functional application and **funcall** (a function) and between the reference to a variable and **function** (a special form). In short, the simultaneous existence of the two worlds and the necessity of interacting between them demand these bridges.

Notice that now it is no longer possible to modify the function environment; there is no assignment form to do so. This property makes it possible for compilers to *inline* function calls in a way that is semantically clean. One of the attractions of multiple name spaces is to be able to give them specific virtues.

2.2.3 Using Lisp_2

To make our definition of Lisp_2 autonomous, we have to indicate what is the global function environment, and how to start the interpreter, **f.evaluate**. The global function environment is coded in a similar way to the environment for variables. We'll change only the macro **defprimitive** to extend one environment but not the other.

```
(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)
```

Now we actually get into that world by this means:

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

2.2.4 Enriching the Function Environment

Environments of any kind are instances of an abstract type. What do we expect from an environment? We expect that it will contain bindings, that we can look there for the binding associated with a name, and that we can also extend it. We

want to have local functions available, and to do so, we want to be able to extend the function environment locally. just as a functional application or the form `let` can extend the environment of variables, At this point, the function environment is frozen, so we would gain a lot by extending it. A new special form, `flet` for *functional let*, will be useful to this purpose. Here's its syntax:

```
(flet ( (name_1 list-of-variables_1 body_1 )
        (name_2 list-of-variables_2 body_2 )
        ...
        (name_n list-of-variables_n body_n ) )
  expressions ... )
```

Since the form `flet` knows how to create only local functions, there is no need to indicate the keyword `lambda` which is implicit. The special form `flet` evaluates the various forms (`lambda` *list-of-variables*$_i$ *body*$_i$) corresponding to the local functions indicated. Then it binds them to the *name*$_i$ in the function environment. The *expressions* forming the body of the `flet` are evaluated in this enriched function environment. All those *name*$_i$ can be used in the function position and can be subjected to `function` if the associated closure is needed in some calculation.

Adding `flet` to `f.evaluate` is straightforward:

```
...
((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) ) ) )) ...
```

Because of `flet`, the possibilities in the function environment increase greatly, and the closure of `fenv` and `env` by the `lambda` form can be explained. For example, consider this:

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

The value of that expression is an anonymous function raising a number to the fourth power. The closure that is created there closes the local function, `square`, and that local function is useful to the closure in its calculations.

2.3 Other Extensions

Once the evaluator has been specialized to handle function terms, new variations come immediately to mind. For example, integers could have a function value assimilating them with list accessors, like this:

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

The integer n is assimilated with the `cad`^{n}`r` if it is positive and with `cd`^{n}`r` if it is negative. The basic accessors, `car` and `cdr`, are `1` and `-1`. Then we can imagine

algebraically rewriting `(-1 (-2 π))` as `(-3 π)` and `(2 (-3 π))` as `(5 π)`.

Another variation could confer a meaning on lists in the function position with the stipulation that they must be lists of functions, like this:

```
((list + - *) 5 3)        → (8 2 15)
```

Applying a list of functions comes down to returning the list of values that each of those functions returns for its arguments. The preceding extract is thus equivalent to this:

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

Finally, we could even allow the function to be in the second position in order to simulate infix notation. In that case, `(1 + 2)` should return `3`. DWIM (that is, Do What I Mean) in [Tei74, Tei76] knows how to recover from that kind of situation.

All these innovations are dangerous because they reduce the number of erroneous forms and thus hide the occurrence of errors that would otherwise be easily detected. Furthermore, they do not lead to any appreciable savings in code, and when everything is taken into account, these innovations are actually rarely used. They also remove that affinity between functions and applicable functional objects, that is, the objects that could appear in the function position. With these innovations, a list or a number would be applicable without so much as becoming a function itself. As a consequence, we could add applicable objects without raising an error, like this:

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

For all those reasons, then, we do not recommend incorporating these innovations into a language such as Lisp. **[see Ex. 2.3]**

2.4 Comparing Lisp_1 and Lisp_2

Now that we are coming to the end of our explorations of Lisp_1 and Lisp_2, what exactly can we say about these two philosophies?

Scheme is a kind of Lisp_1, nice to program and pleasant to teach because the evaluation process is simple and consistent. By comparison, Lisp_2 is more difficult because the existence of the two worlds obliges us to exploit forms that cross over from one world to the other. COMMON LISP is not exactly a Lisp_2 because other binding spaces exist, such as the environment of lexical escapes, the labels of `tagbody` forms, etc. For that reason, we sometimes speak of Lisp_n because we can associate many kinds of behavior with the same name depending on its syntactic position. Languages with strong syntax (indeed, some people would say overwhelming syntax) often have multiple name spaces or multiple environments (environments for variables, for functions, for types, etc.). These multiple environments have specialized properties. If, for example, there are no modifications possible (that is, no assignments) in the local function environment, then it is easy to optimize the call to local functions.

A program written in $Lisp_2$ clearly separates the world of functions from the rest of its computations. This is a profitable distinction that all good Scheme compilers exploit, according to [Sén89]. Internally, these compilers rewrite Scheme programs into a kind of $Lisp_2$ that they can then compile better. They make clear every place where **funcall** has to be inserted, that is, all the calculated calls. A user of $Lisp_2$ has to do much of the work of the compiler in that way and thus understands better what it's worth.

Since we've walked through so many possible variations in the preceding pages, we think it may be useful to give a definition here—the simplest possible—of another instance of $Lisp_2$ inspired by COMMON LISP. The only modification that we're going to include here is to introduce the function **f.lookup** to search for a name in the function environment. If the name cannot be found, a function calling **wrong** will be returned. This device makes it possible to insure that **f.lookup** always returns a function in finite time. Of course, this device also introduces a kind of deferred error since such an error does not occur in the reference to the non-existing function, but rather it occurs in the application of the function, which could occur later or perhaps not at all.

```
(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.eprogn (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) ) ) )
```

Other more pragmatic considerations comparing Lisp_1 and Lisp_2 are connected to readability, according to [GP88]. Surely experienced Lisp programmers avoid writing this kind of code:

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

From the point of view of Lisp_1, `(list list)` is a legal auto-application[2] and its meaning is very different in Lisp_2. In COMMON LISP those two names are evaluated in two different environments, and thus there is no conflict between them. Even so, it is still good programming style to avoid naming local variables with the names of well known global functions; macro-writers will thank you, and your programs will be less dependent on which Lisp or Scheme you happen to use.

Another difference between Lisp_1 and Lisp_2 concerns macros themselves. A macro that expands into a `lambda` form, for example, to implement a system of objects, is highly problematic in COMMON LISP because of the grammatical restrictions that COMMON LISP imposes on the places where `lambda` can appear. A `lambda` form can appear only in the function position, so an expression like this

2. Other auto-applications that make sense do exist, though they are not numerous. Here's another `(number? number?)`.

(... (**lambda** ...) ...) is erroneous. A **lambda** form can also appear in the special form **function**, but that form itself can appear only in the position of a parameter, so this expression ((**function** (**lambda** ...)) ...) is erroneous, too. A macro that is going to be expanded never knows where—in which function context or parametric context—it will be inserted; consequently, a macro cannot be written without inducing a transformation of a program into something more complicated and more global. For that system of objects that we mentioned, we could adopt an expansion toward (**function** (**lambda** ...)) and add **funcall** to the head of all the forms likely to contain an object in the function position.

Finally, we should mention a means that many languages adopt. We can limit the risk of confusion, even with multiple value spaces, by forbidding the same name to appear in more than one space at a time. In the example we looked at earlier, **list** could not be used as a variable because it already appears in the global function environment. Almost all Lisp or Scheme systems also forbid a name to serve simultaneously as the name of a function and of a macro. This kind of rule certainly makes some aspects of life easier.

2.5 Name Spaces

An *environment* associates entities with names. We've already seen two kinds of environments: **env**, the normal environment, and **fenv**, the function environment associating names with functions. The reason we separated those two spaces of values was to improve the way function calls were handled and to distinguish the function world clearly from the variable world. That distinction, however, obliged us to introduce two different evaluators as well as some way of getting from one world to the other—changes that complicate the semantics of the language. When we discussed dynamic variables, we mentioned that recent dialects of Lisp (such as ILOG TALK, EULISP, IS-Lisp) put dynamic variables in a separate name space as well. We're going to look more closely at that variation here. In doing so, we'll illustrate the idea of a *name space*.

An environment is a kind of abstract type. An environment contains *bindings* between names and the *entities* referenced by these names. These entities can be either values (that is, objects that the user can manipulate like any other first class object) or real entities (that is, second class objects that can be handled only by means of their name and generally only across an appropriate set of syntactic elements and special forms). For the moment, the only entities that we recognize are bindings. For us, these entities exist because they can be captured within closures. We'll have more to say about the qualities of these bindings when we study side effects later.

There are many things that we might search for in an environment. We might search to see whether a given name appears in a given environment; we might search for the entity associated with a name; we might search in order to modify that association. We can also extend an environment with new associations (or new bindings) whether that environment is current, local, or global. Of course, not all these operations are necessarily relevant to every environment. In fact, many environments are useful only because they limit such operations. The following

chart, for example, lists the qualities of the environment for variables in Scheme.

Reference	x
Value	x
Modification	`(set!` x ...`)`
Extension	`(lambda (...`x ...`)` ...`)`
Definition	`(define` x ...`)`

We'll be using the ideas in that chart quite frequently to discuss properties of environments, so we'll say a bit more about it now. That first line indicates the syntax we use to reference a variable in a closure. The second line corresponds to the syntax that lets us get the value of a variable. In the case of variables, the syntax for both the value and the closure is the same, but that is not always the case. The third line shows how the binding associated with the variable can be modified. The fourth line shows a way to extend the environment of lexical variables: by a `lambda` form, or of course, by macros such as `let` or `let*` since they expand into `lambda` forms. Finally, the last line of the chart shows how to define a global binding. In case these distinctions seem obscure or feel like overkill to you, we should point out here that the next charts in this chapter will clarify the various intentions behind the ways that variables are used.

In Lisp_2, examined at the beginning of this chapter, the space for functions could be characterized by this chart.

Reference	(*f* ...)
Value	`(function` *f*`)`
Modification	that's not possible here
Extension	`(flet (...(`*f* ...`)` ...`)` ...`)`
Definition	not treated before (cf. `defun`)

2.5.1 Dynamic Variables

Dynamic variables, as a concept, are so different from lexical variables that we're going to treat them separately. The following chart shows the qualities that we want to have in our new environment, the environment for dynamic variables.

Reference	can not be captured
Value	`(dynamic` *d*`)`
Modification	`(dynamic-set!` *d* ...`)`
Extension	`(dynamic-let (` ...`(`*d* ...`)` ...`)` ...`)`
Definition	not treated here.

This new name space takes into account many facts: that dynamic variables[3] can be bound locally by `dynamic-let` with syntax comparable to that of `let` or `flet`; that we can get the value of a dynamic variable by `dynamic`; that we can modify one with `dynamic-set!`.

Those three are special forms that we'll see again later in another implementation. For now, we're going to take the interpreter `f.evaluate` and add a new

3. In this context, "dynamic variable" is a poor choice of name since, in some ways, this is not a specialization of (lexical) variables.

environment to it: **denv**. That new environment will contain only dynamic variables. This new interpreter, which we'll call Lisp_3, will use functions that prefix their names by **df.** to minimize confusion. Here's the new evaluator. (We've removed **flet** to avoid overloading it.)

```
(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))
                (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 ) )
```

Since we introduced the new environment **denv**, we have to augment the signatures of **df.evaluate** and **df.eprogn** to convey that information. In addition, **df.evaluate** introduces three new special forms to handle **denv**, the *dynamic environment*. There's a more subtle modification in **df.evaluate-application**: now a function is applied not only to its arguments but also to the current dynamic environment. You've already seen this situation [see p. 19] when we had to pass the current environment to the invoked function.

With respect to a functional application, there are several environments in play at once. There's the environment for variables and functions that have been captured by the definition of the function. There's also the environment for dynamic variables current at the time of the application. That environment for dynamic variables cannot be captured, and every reference to a dynamic variable involves a search for its value in the current dynamic environment. This search may prove unfruitful; in such a case, we raise an error. Other choices could be possible: for example, to have a unique dynamic global environment, as provided by IS-Lisp; or to have many global dynamic environments, in fact, one per module, as in EuLisp; or even to have the global environment of lexical variables, as in Common Lisp. [see p. 48]

One of the advantages of this new space is that it shows clearly what is dynamic and what isn't. Any time we intervene in the dynamic environment, we do so by means of a special form prefixed by **dynamic**. This eye-catching notation lets us see the interface to a function right away. In practice in this version of Lisp_3, the behavior of a function is stipulated not only by the value of its variables but also by the dynamic environment.

Among the conventional ways of using a dynamic environment, the most important concerns error handling. When an error or some other exceptional situation occurs during a computation, an object defining the exception is formed, and to that object, we apply the current function for handling exceptions (and possibly for recovering from errors). That exception handler could be a global function, but in that case it would require undesirable assignments in order to specify which handler monitors which computation. The "airtightness" of lexical environments hardly lends itself to specifying functions to handle errors. In contrast, a computation has an extent perfectly enclosed by a form like **let** or **dynamic-let**. **dynamic-let** even has several advantages:

1. the bindings that it introduces cannot be captured;
2. those bindings are accessible only during the extent of the computation of its body;
3. those bindings are automatically undone at the end of the computation.

For those reasons, **dynamic-let** is ideal for temporarily establishing a function to handle errors.

Here's another example of how we use dynamic variables. The print functions in COMMON LISP are governed by various dynamic variables such as ***print-base***, ***print-circle***, etc. Those dynamic variables specify such things as the numeric base for printing numbers, whether the data to print entails cycles, and so forth. Of course, it would be possible for every print function to take all this information as arguments. In that case, instead of simply writing (**print** *expression*), we would have to write (**print** *expression print-escape print-radix print-base print-circle print-pretty print-level print-length print-case print-gensym print-array*) each time. That is, dynamic variables are a way of setting parameters for a computation and avoiding exhaustive enumeration of parameters that usually have an acceptable default value.

Scheme, uses a similar mechanism for specifying input and output ports. We can write (**display** *expression*) or[4] (**display** *expression port*). That first form, with only one argument, prints the value of *expression* to the current output port. The second form, with two arguments, specifies explicitly which output port to use. The function **with-output-to-file** lets you specify the current output port during the duration of a computation. You can find out the current output port by means of the function **current-output-port**. We could write a function[5] to print cyclic lists, independent of the print port, in this way:

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

4. One of the democratic principles of Lisp is "Let others do what you allow yourself to do." Notice that the function **display** can take either one or two arguments, but no linguistic mechanism would allow that in Scheme. Lisp, in contrast to Scheme, supports the idea of optional arguments.
5. See also the function **list-length** in COMMON LISP.

If we go back to our chart, we can adapt it to the space for output ports in Scheme, like this:

Reference	referenced every time that it's not mentioned in a print function
Value	`(current-output-port)`
Modification	not modifiable
Extension	`(with-output-to-file` *file-name thunk*`)`
Definition	—not applicable—

In COMMON LISP, this mechanism explicitly uses dynamic variables. By default, the functions `print`, `write`, etc., use the output port with the value of the dynamic variable `*standard-output*`.[6] We could thus simulate[7] `with-output-to-file` by:

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

2.5.2 Dynamic Variables in COMMON LISP

Even though COMMON LISP keeps the concepts of lexical and dynamic variables quite separate, it still tries to unify them syntactically. The form `dynamic-let` doesn't really exist but it could be simulated like this:

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

However, the innovation is that to get the value of the dynamic variable `x` into β, we won't pass by the `dynamic` form, but we'll simply write `x`. The reason: the declaration `(declare (special x))` stipulates two things at once: that the binding that `let` establishes must be dynamic, and that any reference to the name `x` in the body of `let` must be considered equivalent to what we've named `(dynamic x)`.

That strategy is inconvenient because it no longer lets us refer to the lexical variable named `x` inside β; we can only refer to its homonym, the dynamic variable. Going the other direction, we can refer to the dynamic variable `x` in any context by writing `(locally (declare (special x)) x)`. That corresponds to our special form, `dynamic`.

The strategy of COMMON LISP thus lexically specifies the nature of a reference. We can make that mechanism explicit by modifying our interpreter, like this:

```
(define (df.evaluate e env fenv denv)
  (if (atom? e)
      (cond ((symbol? e) (cl.lookup e env denv))
            ((or (number? e)(string? e)(char? e)(boolean? e)(vector? e))
             e )
            (else (wrong "Cannot evaluate" e)) )
      (case (car e)
```

6. It's conventional to put stars around the names of dynamic variables to highlight them.
7. That's not COMMON LISP nor IS-Lisp; it's Lisp_3, as we defined it a little earlier.

```
        ((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!) (cl.update! (cadr e)
                            env
                            denv
                            (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-let)
         (df.eprogn (cddr e)
                    (special-extend env              ;** Modified **
                                    (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)
                ;;lookup in the current dynamic environment
                (let lookup-in-denv ((denv denv))
                  (if (pair? denv)
                      (if (eq? (caar denv) var)
                          (cdar denv)
                          (lookup-in-denv (cdr denv)) )
                      ;;default to the global lexical environment
                      (lookup var env.global) ) )
                (look (cdr env)) ) )
```

```
        (wrong "No such binding" var) ) ) )
```

Here's how that mechanism works: when we bind dynamic variables by **dynamic-let**, we bind them normally in the dynamic environment, but we also mark them as being dynamic in the lexical environment. We code that information by pushing their name into the lexical environment. The process that associates a reference with its value (see function **cl.lookup**) is thus modified, first of all, to achieve the syntactic analysis in order to determine the nature of the reference (whether it is lexical or dynamic); then we search for the associated value in the right environment. In addition, if we do not find the dynamic value, then we go back to the global lexical environment, since it also serves as the global dynamic environment in COMMON LISP.

To give a quick example of how Lisp_3 imitates COMMON LISP in this respect, we could evaluate this:

```
(dynamic-let ((x 2))
   (+ x                          ;dynamic
      (let ((x (+                ;lexical
                  x x )))        ;dynamic
        (+ x                     ;lexical
           (dynamic x) ) ) ) )   ;dynamic
→ 8
```

2.5.3 Dynamic Variables without a Special Form

The way of dealing with dynamic variables that we've presented so far uses three special forms. Since Scheme makes an effort to limit the number of special forms, we might want to consider some other mechanism. Without looking at every conceivable variation that has ever been studied for Scheme, we propose the following because it uses only two functions. The first function associates two values; the second function finds the second of those two values when we hand it the first one. We'll use symbols to name dynamic variables. Finally, if we want to modify those associations, we have to use mutable data, such as a dotted pair. In these ways, we can respect the austerity of Scheme.

As we study this variation, we'll introduce a new interpreter with two environments, **env** and **denv**. This new interpreter is like the previous one except that we have removed a few things from it: all superfluous special forms, the space for functions, and the references to variables, as in COMMON LISP. As a consequence, only the essence of the dynamic environment remains, and that hardly seems useful anymore since it is not modified anywhere. It is, however, always provided to functions, and that provision will enable us to do what we intend. To distinguish this variation from the others, we'll prefix the functions by **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 ) )
```

As we promised, we're going to introduce two functions. We'll name the first of the two **bind-with-dynamic-extent**, and we'll abbreviate that as **bind/de**. As its first parameter, it takes a key, **tag**; as its second, it takes **value** which will be associated with that key; and finally, it takes a **thunk**, that is, a calculation represented by a 0-ary function (that is, a function without variables). The function **bind/de** invokes the thunk after it has enriched the dynamic environment.

```
(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) ) ) )
```

The second function that we'll introduce exploits the dynamic environment. Since we have to make provisions for the case where we do not find what we're looking for in the dynamic environment, the function **assoc/de** takes a key as its first argument, and a function as its second. It will invoke that function on the key if the key is not present in the dynamic environment.

```
(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) ) ) )
```

Many variations on this are possible, depending on whether we use **eqv?** or **equal?** for the comparison. [see **Ex. 2.4**]

To take an earlier example again, we will evaluate this:

```
(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

In that way, we've shown that there is really no need for special forms to get the equivalent of dynamic variables. In doing so, we have actually gained something since we can now associate anything with anything else. Of course, that advantage can be offset by inconvenience since there are efficient implementations of dynamic variables (even without parallelism). For example, shallow binding demands that the value of the key[8] must be a symbol. On the positive side, for this variation, we can count on transparence. Accessing a dynamic variable will surely be expensive because it entails calls to specialized functions. We're rather far from the fusion that COMMON LISP introduced.

Along with all the other inconveniences, we have to mention the syntax that using **bind/de** requires: a thunk and **assoc/de** to handle those associations. Of course, judicious use of macros can hide that problem. Another inconvenience comes when we try to compile calls to these functions correctly. The compiler must know them intimately; it has to know their exact behavior and all their properties. True, the fact that we rarely access this kind of variable and the fact that we have a functional interface to do so both simplify naive compilers.

2.5.4 Conclusions about Name Spaces

At the end of this digression about dynamic variables, it is important to keep in mind the idea of a name space that corresponds to a specialized environment to manipulate certain types of objects. We've seen Lisp_3 at work, and we've looked closely at the method that COMMON LISP uses for dynamic variables.

Nevertheless, that last variation—the one that used only two functions rather than three special forms for the same effects—raises a new problem. If this is a Lisp_n, which n is it? We started off from Scheme, and in the implementation that

8. Many implementations of Lisp forbid the use of keys like `nil` or `if` as the names of variables.

we've given, there are clearly two environments, **env** and **denv**. However, there is only one rule for all evaluations, and that's always consistent with Scheme, and thus it must be a Lisp_1. Yet things are not so clear-cut because we had to modify the definition (just compare the definition of **evaluate** and **dd.evaluate**) in order to implement the functions **bind/de** and **assoc/de**. At that point, we're facing primitive functions that we cannot recreate in pure Scheme if they don't already exist, and moreover the very existence of those two functions in the library of available functions profoundly changes the semantics of the language. In the next chapter, we'll see a similar case with **call/cc**.

In short, we seem to have made a Lisp_1 if we count the number of evaluation rules, but we appear to have a Lisp_2 if we focus on the number of name spaces. A generalization of this observation is that some authorities claim that the existence of a property list is the sign of a Lisp_n, where n is unbounded. Since our definition involves a global name space implemented by non-primitive functions, [**see Ex. 2.6**], we'll settle for that name: Lisp_2.

We can draw another lesson from this study of lexical and dynamic variables. COMMON LISP tries to unify the access to two different spaces by using a uniform syntax, and as a consequence, it has to promulgate rules to determine which name space to use. In COMMON LISP, the dynamic global space is confused with the lexical global space. In Lisp 1.5, there was a concept of constant defined by the special form **csetq** where **c** indicated constant.

Reference	*x*
Value	*x*
Modification	(**csetq** *x form*)
Extension	not possible
Definition	(**csetq** *x form*)

The introduction of constants makes the syntax ambiguous. When we write **foo**, it could be a constant or a variable. The rule in Lisp 1.5 was this: if **foo** is a constant, then return its value; otherwise, search for the value of the lexical variable of the same name. However, constants can be modified (yes! imagine that) by the form that creates them, **csetq**. In that sense, constants belong to the world of global variables in Scheme where the order is the reverse since in Scheme the lexical value is considered first, and the global value is considered only in default of it.

This problem is fairly general. When several different spaces can be referred to by an ambiguous syntax, the rules have to be spelled out in order to eliminate the ambiguity.

2.6 Recursion

Recursion is essential to Lisp, though nothing we've seen yet explains how it is implemented. Now we're going to analyze various kinds of recursion and the different problems that each of them poses.

2.6.1 Simple Recursion

The most widely known simply recursive function is probably the factorial, defined like this:

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

The language that we defined in the previous chapter does not know the meaning of **define**, so let's assume for the moment that this macro expands like this:

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

This prompts us to consider an assignment, that is, a modification impinging on the variable **fact**. Such a modification would make no sense unless the variable exists already. We can characterize the global environment as the place where all variables already exist. That's certainly a kind of *virtual reality* where the effective implementation (and more precisely, the mechanism for reading programs) has to hurry, once a variable is named, in order to create the binding associated with that named variable in the global environment and to pretend that it had always been there. Every variable is thus reputed to be pre-existing: defining a variable is merely a question of modifying its value. This position, however, poses a problem about what can be the value of a variable that has not yet been assigned. The implementation has to make arrangements to trap this new kind of error corresponding to a binding that exists but has not been initialized. Here you can see that the idea of a binding is complicated, so we'll analyze it in greater detail in Chapter 4. [see p. 111]

We can get rid of this problem connected to the troublesome existence of a binding that exists but has not been initialized, if we adopt another point of view. Either assignment can modify only existing bindings, or bindings do not "pre-exist." In that case, it would be necessary to create bindings when we want them, and for that task, **define** could therefore be considered as a new special form with this role. Thus we could not refer to a variable nor modify its value unless it had already been created. Consider the other side of the coin for a moment in the following example.

```
(define (display-pi)
  (display pi) )
(define pi 2.7182818285)                    ; MISTAKE
(define (print-pi)
  (display pi) )
(define pi 3.1415926536)
```

Is that definition of **display-pi** legal? Its body refers to the variable **pi** even though that variable does not yet exist. The fourth definition modifies the second one (to correct it) but even if the meaning of **define** is to create a new binding, does it still have the right to create a new one with the same name?

There is more than one possible response to that rhetorical question. In fact, at least two different positions confront each other here: the first extends the

principle of lexicality to global definitions (ML takes this one); the second, used by Lisp, takes a more dynamic stance.

In a *global* world that is *purely lexical*—which we'll call *hyperstatic*—we cannot talk about a variable (that is, we can't refer to it, nor evaluate it, nor modify it) unless it already exists. In that context, the function `display-pi` is erroneous since it refers to the variable `pi` even though that variable does not yet exist. Any call to the function `display-pi` should surely raise the error "unknown variable: `pi`" if even the system allowed the definition of the `display-pi` function. Such would not be the case of the function `print-pi`; it would display the value that was valid at the time of its definition (in our example, `2.7182818285`), and it would do so indefinitely. In this world, "redefinition" of `pi` is entirely legal, and doing so creates a new variable of the same name having the rest of these interactions as its scope. An imaginative way of seeing this mechanism is to consider the preceding sequence as equivalent to this:

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

The three dots mark the rest of the interactions or the remaining global definitions.

Lisp, as we said, takes a more dynamic view. It assumes that at most only one global variable can exist with a given name, and that this global variable is visible everywhere, and, in particular, Lisp supports forward references to such a variable without the forward reference being highlighted in any syntactic way. (By syntactically highlighted, we mean such conventions as we see in Pascal with `forward` or in ISO C with prototypes as in [ISO90].)

That discussion is non-trivial. When we choose the environment in which the function is evaluated, we have to ask ourselves, "What will be the value of `fact`?" If the evaluation occurs in the global environment, before the binding of `fact` is created, then `fact` cannot be recursive. The reason for that is simple: the reference to `fact` in the body of the function must be looked for in the environment captured by the closure (the one enriched by the closure that has been created), but that environment does not contain the variable `fact`. As a consequence, it is necessary that the environment which will be closed contain not only the global environment but also the variable `fact`. This observation tends to favor a global environment where all the variables pre-exist because then we would not even have to ask that question we began with. However, if we adhere to the second possibility—the hyperstatic vision of a global environment—then we have to be sure that `define` binds `fact` before evaluating the closure that will become the value of this binding. In short, simple recursion demands a global environment.

2.6.2 Mutual Recursion

Now let's suppose that we want to define two mutually recursive functions. Let's say `odd?` and `even?` test (very inefficiently, by the way) the parity of a natural number. Those two are simply defined like this:

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

Regardless of the order in which we put these two definitions, the first one cannot be aware of the second; in this case, for example, **even?** knows nothing about **odd?**. Here again, the global environment, where all variables pre-exist, seems a winner because then the two closures capture the global environment, and thus capture all variables, and thus, in particular, capture **odd?** and **even?**. Of course, we're leaving the details of how to represent a global environment that contains an enumerable number of variables to the implementor.

It is more difficult to adopt the point of view of purely lexical global definitions because the first definition necessarily knows nothing about the second. One solution would be to introduce the two definitions together, at the same time. Doing so would insure their mutual acquaintance with each other. (We'll come back to this point once we have studied local recursion.) For example, if we dig out that old version of **define** from Lisp 1.5, we would write this:

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

Mutual recursion can in that way be expressed by the global environment on condition that it's blessed with *ad hoc* qualities.

What happens now if we want functions that are locally recursive?

2.6.3 Local Recursion in Lisp$_2$

Some of the problems we encountered when we were defining **fact** in a global environment come up again when we want to define **fact** locally. We have to do something so that the call to **fact** in the body of **fact** will be recursive. That is, we need for the function associated with this name to be the factorial value of **fact** in the function environment. However, a major difference exists here: when a binding does not exist in a local environment, an error occurs. Consider what happens if we write this in Lisp$_2$:

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

The function **fact** is bound in the current function environment. This closure captures both the function environment and the parametric environment current in the locality where the form **flet** is evaluated. In consequence, the function **fact** that appears in the body of **fact** refers to the function **fact** valid outside of the form **flet**. There's no reason for that function to be the factorial, and as a consequence, we don't really get recursion here.

That problem had already been recognized in Lisp 1.5 where a special form named **label** made it possible to define a locally recursive function. In that case, we would have written this:

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

```
                                (* n (fact (- n 1))) ) ))
```

That form returns an anonymous function that computes the factorial. Moreover, this anonymous function is the value of the function `fact` that appears in its body.

We cannot guarantee, however, that Lisp 1.5 was an authentic Lisp_2, and as interesting as it is, the form `label`, unfortunately, is not able to handle the case of mutual recursion simply. For that reason, an n-ary version was invented much later, according to [HS75]: the special form `labels`. This later form has the same syntax as the form `flet`, but it insures that the closures that are created will be in a function environment where all the local functions are aware of each other. Thus we can also locally define the recursive function `fact` as well as the mutually recursive functions `even?` and `odd?`, as we show in this:

```
(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
```

In Lisp_2, then we have two forms available, `flet` and `labels`, to enrich the local function environment.

2.6.4 Local Recursion in Lisp_1

The problem of defining locally recursive functions occurs in Lisp_1 as well, and we solve that problem in a similar way. A particular form, known as `letrec` for "let recursive," has much the same effect as `labels`.

In Scheme, `let` has the following syntax:

```
(let ((variable1 expression1)
      (variable2 expression2)
      ...
      (variablen expressionn) )
  expressions... )
```

Its effect is equivalent to this expression:

```
((lambda (variable1 variable2 ... variablen) expressions...)
 expression1 expression2 ... expressionn )
```

Here's a more discursive explanation of what's going on. First, the expressions, $expression_1$, $expression_2$, ... $expression_n$, are evaluated; then the variables, $variable_1$, $variable_2$, ... $variable_n$, are bound to the values that have already been gotten; finally, in the extended environment, the body of `let` is evaluated (within an implicit `begin`), and its value becomes the value of the entire `let` form.

A priori, the form `let` does not seem useful since it can be simulated by `lambda` and thus can be only a simple macro. (By the way, this is not a special form in Scheme, but primitive syntax.) Nevertheless, the form `let` becomes very useful on the stylistic level because it allows us to put a variable just next to its initial value, like a block in Algol. This idea leads us to remark that the variables in a `let` form

are initialized by values computed in the current environment; only the body of `let` is evaluated in the enriched environment.

For the same reasons that we encountered in Lisp_2, this way of doing things means that we cannot write mutually recursive functions in a simple way, so we'll use `letrec` again here for the same purpose.

The syntax of the form `letrec` is similar to that of the form `let`. For example, we would write:

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

The difference between `letrec` and `let` is that the expressions initializing the variables are evaluated in the same environment where the body of the `letrec` form is evaluated. The operations carried out by `letrec` are comparable to those of `let`, but they are not done in the same order. First, the current environment is extended by the variables of `letrec`. Then, in that extended environment, the initializing expressions of those same variables are evaluated. Finally, the body is evaluated, still in that enriched environment. This description of what's going on suggests clearly how to implement `letrec`. To get the same effect, we simply have to write this:

```
(let ((even? 'void)
      (odd? 'void) )
   (set! even? (lambda (n) (or (= n 0) (odd? (- n 1)))))
   (set! odd? (lambda (n) (or (= n 1) (even? (- n 1)))))
   (even? 4) )
```

The bindings for `even?` and `odd?` are created. (By the way, the variables are bound to values of no particular importance because `let` or `lambda` do not allow uninitialized bindings to be created.) Then those two variables are initialized with values computed in an environment that is aware of the variables `even?` and `odd?`. We've used the phrase "aware of" because, even though the bindings of the variables `even?` and `odd?` exist, their values have no relation to what we expect from them inasmuch as they have not been initialized. The bindings of `even?` and `odd?` exist enough to be captured but not enough to be evaluated validly.

However, the transformation is not quite correct because of the issue of order: a `let` form is equivalent to a functional application, and its initialization forms become the arguments of the functional application that's being generated; if indeed a `let` form is equivalent to a functional application, then the order of evaluating the initialization forms should not be specified. Unfortunately, the expansion we just used forces a particular order: left to right. **[see Ex. 2.9]**

Equations and `letrec`

A major problem of `letrec` is that its syntax is not very strict; in fact, it allows anything as initialization forms, and not solely functions. In contrast, the syntax of `labels` in COMMON LISP forbids the definition of anything other than functions. In Scheme, it's possible to write this:

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

Notice that the variable **x** is defined in terms of itself according to the semantics of `letrec`. That's a veritable equation written like this:

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

It seems logical to bind **x** to the solution of this equation, so the final value of that `letrec` form would be `1`.

But what happens according to this interpretation when an equation has no solution or has multiple solutions?

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

There are many other domains (all familiar in Lisp) like S-expressions, where we can sometimes insure that there will always be a unique solution, according to [MS80]. For example, we can build an infinite list without apparent side effects, rather like a language with lazy evaluation, by writing this:

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

The value of that form would then be either the infinite list `(bar bar bar ... )` calculated the lazy way, as in [FW76, PJ87], or the circular structure (less expensive) that we get from this:

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

For that reason, we have to adopt a more pragmatic rule for `letrec` to forbid the use of the value of a variable of `letrec` during the initialization of the same variable. Accordingly, the two preceding examples demand that the value of **x** must already be known in order to initialize **x**. That situation raises an error and will be punished. However, we should note that the order of initialization is not specified in Scheme, so certain programs can be error-prone in certain implementations and not so in others. That's the case, for example, with this:

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

If the initialization form of **y** is evaluated before the one for **x**, then everything turns out fine. In the opposite case, there will be an error because we have to try to increment **y** which is bound but which does not yet have a value. Some Scheme or ML compilers analyze initialization expressions and sort them topologically to determine the order in which to evaluate them. This kind of sorting is not always feasible when we introduce mutual dependence[9] like this:

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

That example reminds us strongly about that discussion of the global environment and the semantics of `define`. There we had a problem of the same kind: how to know about the existence of an uninitialized binding.

9. Again, returning the list `(42 42)` does not violate the statement of this form.

2.6.5 Creating Uninitialized Bindings

The official semantics of Scheme makes `letrec` derived syntax, that is, a useful abbreviation but not really necessary. Accordingly, every `letrec` form is equivalent to a form that could have been written directly in Scheme. We tried that approach earlier when we bound the variables of `letrec` temporarily to `void`. Unfortunately, doing so initialized the bindings and made it impossible to detect the error we mentioned before. Our misfortune is actually even worse than that because none of the four special forms in Scheme allows the creation of uninitialized bindings.

A first attempt at a solution might use the object `#<UFO>` [see p. 14] in place of `void`. Of course, we can't do much with `#<UFO>`: we can't add it, nor take its `car`, but since it is a first class object, we can make it appear in a `cons`, so the following program would not be false and would return `#<UFO>`:

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

The underlying reason for that is that the lack of initialization of a binding is a property of the binding itself, not of its contents. Consequently, using a first class value, `#<UFO>`, is not a solution to our problem.

If we think about implementations, then we notice that often a binding is uninitialized if it has a very particular value. Let's call that very special value `#<uninitialized>`, and let's suppose for the moment that this value is first class. When it appears as the value of a variable, then the variable has to be regarded as uninitialized. We will thus replace `void` by `#<uninitialized>` and get the error detection that we wanted. However, this mechanism is too powerful because anybody can provide `#<uninitialized>` as an argument to any function, and we thus lose an important property that every variable of a function has a value. According to those terms, our old reliable factorial function cannot even assume any longer that its variable `n` is bound and thus it must test explicitly whether `n` has a value, like this:

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

This surcharge is too costly and, consequently, `#<uninitialized>` can't be a first class value; it can be only an internal flag reserved to the implementor, one that the end-user can't touch. This second path is consequently closed to us, as far as a solution to our problem goes.

A third solution is to introduce a new special form, capable of creating uninitialized bindings. Let's take advantage of a syntactic variation of `let`, one that exists in COMMON LISP but not in Scheme, to do that, like this:

```
(let ( variable ... )
     ... )
```

When a variable appears alone, without an initialization form, in the list of local variables in a `let` form, then the binding will be created uninitialized. Any evaluation, or even any attempt at evaluation, of this variable must verify whether it is really initialized. The expansion of a `letrec` now no longer calls anything

foreign to it. In what follows, the variables $temp_i$ are *hygienic*; that is, they cannot provoke conflicts with the $variables_i$ nor with those that are free in π.

```
(letrec ((variable1 expression1)
         ...
         (variablen expressionn) )
   body )
```

≡

```
(let ( variable1 ... variablen )
  (let (( temp1 expression1 )
        ...
        ( tempn expressionn ) )
    (set! variable1 temp1)
    ...
    (set! variablen tempn)
    body ) )
```

In that way, we've resolved the problem in a satisfactory way because only the uninitialized variables pay the surcharge due to uninitialization. However, the form `let` is no longer just syntax; rather, it is a primitive special form that must thus appear in the basic interpreter. For that reason, we'll add the following clause to `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) ) ) ) ) ...
```

The variable `the-uninitialized-marker` belongs to the definition language. It could be defined like this:

```
(define the-non-initialized-marker (cons 'non 'initialized))
```

Of course, the internal flag is exploited by the function `lookup` which must be adapted for it. The function `update!` is not affected by this change. In the following function, the two different calls to `wrong` characterize two different situations: non-existing binding and uninitialized binding.

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

After these syntactic-semantic ramblings, we have a form of `letrec` that allows us to co-define local functions that are mutually recursive.

2.6.6 Recursion without Assignment

The form **letrec** that we've been analyzing uses assignments to insure that initialization forms are evaluated. Languages that are known as *purely functional* don't have this resource available to them; side effects are unknown among them, and what is assignment if not a side effect on the value of a variable?

As a philosophy, forbidding assignment offers great advantages: it preserves the referential integrity of the language and thus leaves an open field for many transformations of programs, such as moving code, using parallel evaluation, using lazy evaluation, etc. However, if side effects are not available, certain algorithms are no longer so clear, and the introspection of a real machine is impeded, since real machines work only because of the side effects of their instructions.

The first solution that comes to mind is to make **letrec** another special form, as it is in most languages in the same class as ML. We would enrich **evaluate** so that it could handle the case of **letrec**, like this:

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

In that way, we regain side effects, formerly attributed to assignments, now carried out by **update!**. We also note that the order of evaluation is still not significant because **map** (in contrast to **for-each**) does not guarantee anything about the order in which it handles terms in a list.[10]

letrec and the Purely Lexical Global Environment

A purely lexical global environment allows the use of a variable only if the variable has already been defined. The problem with that rule is that we cannot define simply recursive functions nor groups of mutually recursive functions. With **letrec**, we can suggest how to resolve that problem by making it possible to define more than one function at a time by indicating whether the definition is recursive or not. We would thus write this:

```
(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))))) )
     ...
```

Those dots represent the rest of the interactions or definitions.

10. The cost here is that we have to build a useless list—useless since it is forgotten as soon as it is built.

Paradoxical Combinator

If you have experience with λ-calculus, then you probably remember how to write *fixed-point combinators* and among them, the famous *paradoxical combinator*, **Y**. A function f has a fixed point if there exists an element x in its domain such that $x = f(x)$. The combinator **Y** can take any function of λ-calculus and return a fixed point for it. This idea is expressed in one of the loveliest and most profound theorems of λ-calculus:

Fixed Point Theorem: $\exists \mathbf{Y}, \forall F,\ \mathbf{Y}F = F(\mathbf{Y}F)$

In Lisp terminology, **Y** is the value of this:

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

Our demonstration of that assertion is quite short. If we assume that **Y** is equal to **(WW)**, then how should we choose **W** so that **(WW)F** will be equal to **F((WW)F)**? Obviously, **W** is no other than $\lambda W.\lambda F.F((WW)F)$. That Lisp expression is nothing other than a transcription of these ideas.

The problem here is that the strategy of call by value in Scheme is not compatible with this kind of programming. We are obliged to add a superfluous η-conversion (superfluous from the point of view of pure λ-calculus) to block any premature evaluation of the term `((w w) f)`. We'll cut across then to the following fixed-point combinator in which `lambda (x) ( ... x))` marks the η-conversion.

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

The most troubling aspect of that definition is how it works. (We're going to show you that right away.) Let's define the function `meta-fact` like this:

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

That function has a disconcerting relation to factorial. We will verify by example that the expression `(meta-fact fact)` calculates the factorial of a number just as well as `fact`, though more slowly. More precisely, let's suppose that we know a fixed point f of `meta-fact`; in other words, f =`(meta-fact` f`)`. That fixed point has, by definition, the property of being a solution to the functional equation in f:

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

So what is f? It can't be anything other than our well known factorial.

Actually, nothing guarantees that the preceding functional equation has a solution, nor that it is unique. (All those terms, of course, must be mathematically

well defined, though that is beyond the purpose of this book.) Indeed, the equation has many solutions, for example:

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

We urge you to verify that the function **another-fact** is yet another fixed point of **meta-fact**. Analyzing all these fixed points shows us that there is a set of answers on which they all agree: they all compute the factorial of natural numbers. They diverge from one another only where **fact** diverges in infinite calculations. For negative integers, **another-fact** returns one result where many might be possible since the functional equation says nothing[11] about those cases. Therefore, there must exist a least fixed point which is the least determined of the solution functions of the functional equation.

The meaning to attribute to a definition like the one for **fact** in the global environment is that it defines a function equal to the least fixed point of the associated functional equation, so when we write this:

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

we should realize that we have just written an equation where the variable is named **fact**. As for the form **define**, it resolves the equation and binds the variable **fact** to that solution of the functional equation. This way of looking at things takes us far from discussions about initialization of global variables [see p. 54] and turns **define** into an equation-solver. Actually, **define** is implemented as we suggested before. Recursion across the global environment coupled with normal evaluation rules does, indeed, compute the least fixed point.

Now let's get back to **fix**, our fixed-point combinator, and trace the execution of **((fix meta-fact) 3)**. In the following trace, remember that no side effects occur, so we'll substitute their values for certain variables directly from time to time.

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

11. See [Man74] for more thorough explanations.

```
    meta-fact )
   3 )
= ((meta-fact (lambda (x)
                (((w w) meta-fact) 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)                    ;(ii)
           (f (lambda (x)
                (((w w) f) x) )) )|
                                  | w≡ (lambda (w)
                                         (lambda (f)
                                           (f (lambda (x)
                                                (((w w) f) x) )) ) )
         meta-fact )
        2 ) )
```

We'll pause there to note that, in gross terms, the expression at step i occurs again at step ii, and, like the thread in a needle, it leads to this:

```
(* 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
```

Notice that as the computation winds its way along, an object corresponding to the factorial has, indeed, been constructed. It's this value:

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

The cleverness lies in the way we recompose a new instance of this factorial every time a recursive call needs it.

That's the way that we could get simple recursion, without side effects, by means of **fix**, a fixed-point combinator. Thanks to **Y** (or to **fix** in Lisp), it is possible to define **define** as a solver of recursive equations; it takes an equation as its argument, and it binds the solution to a name. Thus the equation defining the factorial leads to binding the variable **fact** to this value:

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

We can extend this technique to the case of multiple functions that are mutually recursive by regrouping them. The functions **odd?** and **even?** can be fused, like this:

```
(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))
```

The problem with this method, however, is that it is not very efficient in terms of performance, compared to even an imperfect compilation of a `letrec` form. (Yet see [Roz92, Ser93].) Nevertheless, this method has its own practitioners, especially when it's time to write books. Functional languages do not adopt this method either, according to [PJ87], because of its inefficiency and because the definition of `fix` is not susceptible to typing. In effect, its nature is to take a functional[12] that takes, as its argument, the function typed $\alpha \rightarrow \beta$ and that returns a fixed point of that functional. The type is consequently this:

`fix:` $((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \beta)$

But in the definition of `fix`, we have the auto-application `(d d)` that has γ for its type, like this:

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

We need a system of non-trivial types to contain this recursive type, or we have to consider `fix` a primitive function that (fortunately) already exists because we don't know how to define it within the language, if it's missing.

2.7 Conclusions

This chapter has crossed several of the great chasms that split the Lisp world in the past thirty years. When we examine the points where these divergences begin, though, we see that they are not so grand. They generally have to do with the meaning of `lambda` and the way that a functional application is calculated. Even though the idea of a function seems like a well founded mathematical concept, its incarnation within a functional language (that is, a language that actually uses functions) like Lisp is quite often the source of many controversies. Being aware of and appreciating these points of divergence is part of Lisp culture. When we analyze the traces of our elders in this culture, we not only build up a basis for mutual understanding, but we also improve our programming style.

Paradoxically, this chapter has shown the importance of the idea of binding. In Lisp_1, a variable (that is, a name) is associated with a unique binding (possibly global) and thus with a unique value. For that reason, we talk about the value of a variable rather than the value associated with the binding of that variable. If we look at a binding as an abstract type, we can say that a binding is created by a binding form, and that it is read or written by evaluation of the variable or by

12. McCarthy in [MAE+62] defines a functional as a function that can have functions as arguments.

assignment, and finally, that it can be captured when a closure is created in which the body refers to the variable associated with that binding.

Bindings are not first class objects. They are handled only indirectly by the variables with which they are associated. Nevertheless, they have an indefinite extent; in fact, they are so useful because they endure.

A binding form introduces the idea of scope. The scope of a variable or of a binding is the textual space where that variable is visible. The scope of a variable bound by `lambda` is restricted to the body of that `lambda` form. For that reason, we talk about its textual or lexical scope.

The idea of binding is complicated by the fact of assignment, and we'll study it in greater detail in the next chapter.

2.8 Exercises

Exercise 2.1 : The following expression is written in COMMON LISP. How would you translate it into Scheme?

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

Exercise 2.2 : In the pseudo-COMMON LISP you saw in this chapter, what is the value of this program? What does it make you think of?

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

Exercise 2.3 : Incorporate the first two innovations presented in Section 2.3 [see p. 39] into the Scheme interpreter. Those innovations concern numbers and lists in the function position.

Exercise 2.4 : The function `assoc/de` could be improved to take a comparer (such as `eq?`, `equal?`, or others) as an argument. Write this new version.

Exercise 2.5 : With the aid of `bind/de` and `assoc/de`, write macros that simulate the special forms `dynamic-let`, `dynamic`, and `dynamic-set!`.

Exercise 2.6 : Write the functions `getprop` and `putprop` to simulate lists of properties. We could associate a value with the key in the list of properties of a symbol with `putprop`; we could search the list of properties for the value associated with a key with `getprop`. Of course, we have the following property:

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

Exercise 2.7 : Define the special form `label` in Lisp_1.

Exercise 2.8 : Define the special form `labels` in Lisp_2.

Exercise 2.9 : Think of a way to expand `letrec` in terms of `let` and `set!` so that the order does not matter when initializations of expressions are evaluated.

Exercise 2.10 : The fixed-point combinator in Scheme has a weakness: it works only for unary functions. Give a new version, `fix2`, that works correctly for binary functions. Then write a version, `fixN`, that works correctly for functions of any arity.

Exercise 2.11 : Now write a function, `NfixN`, that returns a fixed point from a list of functionals of any arity. We would use such a thing, for example, to define this:

```
(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)) )
```

Exercise 2.12 : Here's the function `klop`. Is it a fixed-point combinator? Try to show whether or not (`klop` f) returns a fixed-point of f, the way `fix` would do.

```
(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) ) )
```

Exercise 2.13 : If the function `hyper-fact` is defined like this:

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

then what is the value of `((hyper-fact hyper-fact) 5)`?

Recommended Reading

In addition to the paper about λ-calculus [SS78a] that we mentioned earlier, you could also consider the analysis of functions in [Mos70] and the comparative study of Lisp_1 and Lisp_2 in [GP88].

There is an interesting introduction to λ-calculus in [Gor88].

The combinator **Y** is also discussed in [Gab88].

3
Escape & Return: Continuations

EVERY computation has the goal of returning a value to a certain entity that we call a *continuation*. This chapter explains that idea and its historic roots. We'll also define a new interpreter, one that makes continuations explicit. In doing so, we'll present various implementations in Lisp and Scheme and we'll go into greater depth about the programming style known as "Continuation Passing Style." Lisp is distinctive among programming languages because of its elaborate forms for manipulating execution control. In some respects, that richness in Lisp will make this chapter seem like an enormous catalogue [Moz87] where you'll probably feel like you've seen a thousand and three control forms one by one. In other respects, however, we'll keep a veil over continuations, at least over how they are physically carried out. Our new interpreter will use objects to show the relatives of continuations and its control blocks in the *evaluation stack*.

The interpreters that we built in earlier chapters took an expression and an environment in order to determine the value of the expression. However, those interpreters were not capable of defining computations that included *escapes*, useful control structures that involve getting out of one context in order to get into another, more preferable one. In conventional programming, we use escapes principally to master the behavior of programs in case of unexpected errors, or to program by exceptions when we define a general behavior where the occurrence of a particular event interrupts the current calculation and sends it back to an appropriate place.

The historic roots of escapes go all the way back to Lisp 1.5 to the form **prog**. Though that form is now obsolete, it was originally introduced in the vain hope of attracting Algol users to Lisp since it was widely believed that they knew how to program only with **goto**. Instead, it seems that the form distracted otherwise healthy Lisp programmers away from the precepts[1] of tail-recursive functional programming. Nevertheless, as a form, **prog** merits attention because it embodies several interesting traits. Here, for example, is factorial written with **prog**.

1. As an example, consider the stylistic differences between the first and third editions of [WH88].

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) ) )
```

The special form **prog** first declares the local variables that it uses (here, only the variable **r**). The expressions that follow are instructions (represented as lists) or labels (represented by symbols). The instructions are evaluated sequentially like in **progn**; and normally the value of a **prog** form is **nil**. Multiple special instructions can be used in one **prog**. Unconditional jumps are specified by **go** (with a label which is not computed) whereas **return** imposes the final value of the **prog** form. There was one restriction in Lisp 1.5: **go** forms and **return** forms could appear only in the first level of a **prog** or in a **cond** at the first level.

The form **return** made it possible to get out of the **prog** form by imposing its result. The restriction in Lisp 1.5 allowed only an escape from a simple context; successive versions improved this point by allowing **return** to be used with no restrictions. Escapes then became the normal way of taking care of errors. When an error occurred, the calculation tried to escape from the error context in order to regain a safer context. With that point in mind, we could rewrite the previous example in the following equivalent code, where the form **return** no longer occurs on the first level.

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) ) )
```

If we reduce the forms **return** and **prog** to nothing more than their control effects, we see clearly that they handle the calling point (and the return point) of the form **prog** exactly like in the case of a functional application where the invoked function returns its value precisely to the place where it was called. In some sense, then, we can say that **prog** binds **return** to the precise point that it must come back to with a value. Escape, then, consists of not knowing the place from which we take off while specifying the place where we want to arrive. Additionally, we hope that such a *jump* will be implemented efficiently: escape then becomes a programming method with its own disciples. You can see it in the following example of searching for a symbol in a binary tree. A naive programming style for this algorithm in Scheme would look like this:

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

Let's assume that we're looking for the symbol `foo` in the expression `(((a . b) . (foo . c)) . (d . e))`. Since the search takes place from left to right and depth-first, once the symbol has been found, then the value `#t` must be passed across several embedded `or` forms before it becomes the final value (the whole point of the search, after all). The following shows the details of the preceding calculation in equivalent forms.

```
(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
```

An efficient escape seems appropriate there. As soon as the symbol that is being searched for has been found, an efficient escape has to support the return of the very same value without considering any remaining branches of the search tree.

Another example comes from programming by exception where repetitive treatment is applied over and over again until an exceptional situation is detected in which case an escape is carried out in order to *escape* from the repetitive treatment that would otherwise continue more or less perpetually. We'll see an example of this later with the function `better-map`. [see p. 80]

If we want to characterize the entity corresponding to a calling point better, we note that a calculation specifies not only the expression to compute and the environment of variables in which to compute it, but also where we must return the value obtained. This "where" is known as a *continuation*. It represents all that remains to compute.

Every computation has a continuation. For example, in the expression `(+ 3 (* 2 4))`, the continuation of the subexpression `(* 2 4)` is the computation of an addition where the first argument is `3` and the second is expected. At this point, theory comes to our rescue: this continuation can be represented in a form

that's easier to see—as a function. A continuation represents a computation, and it doesn't take place unless we get a value for it. This protocol strongly resembles the one for a function and is thus in that guise that continuations will be represented. For the preceding example, the continuation of the subexpression `(* 2 4)` will thus be equivalent to `(lambda (x) (+ 3 x))`, thus faithfully underlining the fact that the calculation waits for the second argument in the addition.

Another representation also exists; it specifies continuations by contexts inspired from λ-calculus. In that representation, we would denote the preceding continuation as `(+ 3 [])`, where `[]` stands for the place where the value is expected.

Truly everything has a continuation. The evaluation of the condition of an alternative is carried out by means of a continuation that exploits the value that will be returned in order to decide whether to take the true or false branch of the alternative. In the expression `(if (foo) 1 2)`, the continuation of the functional application `(foo)` is thus `(lambda (x) (if x 1 2))` or `(if [] 1 2)`.

Escapes, programming by exception, and so forth are merely particular forms for manipulating continuations. With that idea in mind, now we'll detail the diverse forms and great variety of continuations observed over the past twenty years or so.

3.1 Forms for Handling Continuations

Capturing a continuation makes it possible to handle the control thread in a program. The form **prog** already makes it possible to do that, but carries with it other effects, like those of a **let** for its local variables. Paring down **prog** so that it concerns only control was the first goal of the forms **catch** and **throw**.

3.1.1 The Pair `catch/throw`

The special form **catch** has the following syntax:

`(catch` *label forms...*`)`

The first argument, *label*, is evaluated, and its value is associated with the current continuation. This facts makes us suppose that there must be a new space, the *dynamic escape space*, binding values and continuations. If we can't think of labels which are not identifiers, we actually can't talk anymore about a name space; this, however, really is one, if we accept the fact that every value can be a valid label for that space. Yet that condition poses the problem of equality within this space: how can we recognize that one value names what another has associated with a continuation?

The other *forms* make up the body of the form **catch** and are themselves evaluated sequentially, like in a **progn** or in a **begin**. If nothing happens, the value of the form **catch** is that of the last of the forms in its body. However, what can intervene is the evaluation of a **throw**.

The form **throw** has the following syntax.

`(throw` *label form*`)`

The first argument is evaluated and must lead to a value that a dynamically embedding **catch** has associated with a continuation. If such is the case, then the

evaluation of the body of this **catch** form will be interrupted, and the value of the *form* will become the value of the entire **catch** form.

Let's go back to our example of searching for a symbol in a binary tree, and let's put into it the forms **catch** and **throw**. The variation that you are about to see has factored out the search process in order not to transmit the variable **id** pointlessly, since it is lexically visible in the entire search. We will thus use a local function to do that.

```
(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)) )
```

As its name indicates, **catch** traps the value that **throw** sends it. An escape is consequently a direct transmission of the value associated with a control manipulation. In other words, the form **catch** is a binding form that associates the current continuation with a label. The form **throw** actually makes the reference to this binding. That form also changes the thread of control as well; **throw** does not really have a value because it returns no result by itself, but it organizes things in such a way that **catch** can return a value. Here, the form **catch** captures the continuation of the call to **find-symbol**, while **throw** accomplishes the direct return to the caller of **find-symbol**.

The dynamic escape space can be characterized by a chart listing its various properties.

Reference	(**throw** *label* ...)
Value	no because it's a second class continuation
Modification	no
Extension	(**catch** *label* ...)
Definition	no

As we said earlier, **catch** is not really a function, but a special form that successively evaluates its first argument (the label), binds its own continuation to this latter value in the dynamic escape environment, and then begins the evaluation of its other arguments. Not all of them will necessarily be evaluated. When **catch** returns a value or when we escape from it, **catch** dissociates the label from the continuation.

The effect of **throw** can be accomplished either by a function or by a special form. When it is a special form, as it is in COMMON LISP, it calculates the label, it verifies the existence of an associated **catch**, then it evaluates the value to transmit, and finally it jumps. When **throw** is a function, it does things in a different order: it first calculates the two arguments, then verifies the existence of an associated **catch**, and then it jumps.

These semantic differences clearly show how unreliably a text might describe the behavior of these control forms. A lot of questions remain unanswered. What happens, for example, when there is no associated **catch**? Which equality is used to compare labels? What does this expression do (**throw** α (**throw** β π))? We'll find answers to those kinds of questions a little later.

3.1.2 The Pair `block/return-from`

The escapes that `catch` and `throw` perform are dynamic. When `throw` requests an escape, it must verify during execution whether an associated `catch` exists, and it must also determine which continuation it refers to. In terms of implementation, there is a non-negligible cost here that we might hope to reduce by means of *lexical* escapes, as they are known in the terminology of COMMON LISP. The special forms `block` and `return-from` superficially resemble `catch` and `throw`.

Here's the syntax of the special form `block`:

`(block` *label forms...*`)`

The first argument is not evaluated and must be a name suitable for the escape: an identifier. The form `block` binds the current continuation to the name *label* within the *lexical escape environment.* The body of `block` is evaluated then in an implicit `progn` with the value of this `progn` becoming the value of `block`. This sequential evaluation can be interrupted by `return-from`.

Here's the syntax of the special form `return-from`:

`(return-from` *label form*`)`

That first argument is not evaluated and must mention the name of an escape that is lexically visible; that is, a `return-from` form can appear only in the body of an associated `block`, just as a variable can appear only in the body of an associated `lambda` form. When a `return-from` form is evaluated, it makes the associated `block` return the value of *form.*

Lexical escapes generate a new name space, and we'll summarize its properties in the following chart.

Reference	`(return-from` *label* ...`)`
Value	no because it's a second class continuation
Modification	no
Extension	`(block` *label* ...`)`
Definition	no

Let's look again at the preceding example, this time, writing[2] it like this:

```
(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) ) ) )
```

Notice that this is not an ordinary translation of the previous example where "`catch 'find`" simply turns into "`block find`" nor likewise for `throw`. Without the migration of "`block find`," the form `return-from` would not have been bound to the appropriate `block`.

2. Warning to Scheme-users: `define` is translated internally into a `letrec` because `(let () (define ...) ...)` is not valid in Scheme because of the `()`.

The generated code corresponding to that form is highly efficient. In gross, general terms, **block** stores the height of the execution stack under the name of the escape **find**. The form **return-from** consists solely of putting **#t** where a value is expected (for example, in a register) and re-adjusting the stack pointer to the height that **block** associated with **find**. This represents only a few instructions, in contrast to the dynamic behavior of **catch**. **catch** would work through the entire list of valid escapes, little by little, until it found one with the right label. You can see that difference clearly if you consider this simulation of **catch**[3] by **block**.

```
(define *active-catchers* '())
(define-syntax throw
  (syntax-rules ()
    ((throw tag value)
     (let* ((label tag)                                    ;compute once
            (escape (assv label *active-catchers*)) )      ;compare with eqv?
       (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 ) ) ))
```

In that simulation, the cost of the pair **catch/throw** is practically entirely concentrated in the call to **assv**[4] in the expansion of **throw**. But first, let's explain how the simulation works. A global variable (here, named ***active-catchers***) keeps track of all the active **catch** forms, that is, those whose execution has not yet been completed. That variable is updated at the exit from **catch** and consequently at the exit from **throw**. The value of ***active-catchers*** is an A-list, where the keys are the labels of **catch** and the values are the associated continuations. This A-list embodies the dynamic escape environment for which **catch** was the binding form and for which **throw** was the referencing form, as you can easily see in that simulation code.

3.1.3 Escapes with a Dynamic Extent

That simulation, however, is imperfect in the sense that it prohibits simultaneous uses of **block**; doing so would perturb the value of the variable ***active-catchers***.

3. The definition of **catch** uses a **block** with the name **label**. The rules for hygienic naming of variables, of course, should be extended to the name space.
4. In passing, you see that the labels are compared by **eqv?**.

The art of simulation or syntactic extension of a dialect is a difficult one, as [Fel90, Bak92c] observe, because frequently adding new traits requires a complicated architecture that prohibits direct access to resources mobilized for the simulation or extension. Later, we'll show you an authentic simulation of **catch** by **block**, but it will require yet another linguistic means: **unwind-protect**.[5]

Like all entities in Lisp, continuations have a certain extent. In the simulation of **catch** by **block**, you saw that the extent of a continuation caught by **catch** is limited to the duration of the calculation in the body of the **catch** in question. We refer to this extent as *dynamic*. When we use that term, we are also thinking of dynamic binding since it insures only that the value of a dynamic variable lasts as long as the evaluation of the body of the corresponding binding form. We could take advantage of this property to offer a new definition of **catch** and **throw** in terms of **block** and **return-from**; this time, the list of catchers that can be activated is easily maintained in the dynamic variable ***active-catchers***. This program reconciles **block** and **catch** so that they can be used simultaneously now.

```
(define-syntax throw
  (syntax-rules()
    ((throw tag value)
     (let* ((label tag)                                    ;compute once
            (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 ) ) ) ) )
```

The extent of an escape caught by **block** is dynamic in COMMON LISP, so the escape can be used only during the calculation of the body of the **block**. Likewise, the extent of an escape caught by **catch** is also dynamic in COMMON LISP, so again, the escape can be used only during the calculation of the body of the **catch**. This fact poses an interesting problem with **block**, a problem that does not occur with **catch**: if **throw** and **return-from** make it possible to abbreviate computations, then some computation must exist to be escaped from. Consider the following program.

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

The value of the first term is an escape to **foo**, but this escape is obsolete when it is applied, and in consequence, we get an execution error. When the closure

5. Another solution would be to redefine the new forms **block** and **return-from**, with the help of the old ones, so that they would be compatible with **catch** and **throw**. That solution is difficult to implement directly, but it can be done by adding a new level of interpretation.

is created, it closes the lexical escape environment, especially the binding of **foo**. That closure is then returned as the value of the form **block**, but that return occurs when we exit from **block**, and consequently, it is out of the question to exit yet again, since that has already happened. When that closure is applied, we verify whether the continuation associated with **foo** is still valid, and we jump if that is the case. The following example shows that closures around lexical escapes do not mix bindings.

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

Compare those results with what we would get by replacing "**block foo**" by "**catch 'foo**" like this:

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

The catcher invoked by the function **f1** is the more recent one, having bound the label **foo**; the result of **catch** is consequently multiplied by **2** and returns **2** finally.

3.1.4 Comparing `catch` and `block`

The forms **catch** and **block** have many points of comparison. The continuations that they capture have a dynamic extent: they last only as long as an evaluation. In contrast, **return-from** can refer an indefinitely long time to a continuation, whereas **throw** is more limited. In terms of efficiency, **block** is better in most cases because it never has to verify during a **return-from** whether the corresponding **block** exists, since that existence is guaranteed by the syntax. However, it is necessary to verify that the escape is not obsolete, though that fact is often syntactically visible. You can see a parallel between dynamic and lexical escapes, on one side, and dynamic and lexical variables, on the other: many problems are common to both groups.

Dynamic escapes allow conflicts that are completely unknown to lexical escapes. For one thing, dynamic escapes can be used anywhere, so one function can put an escape in place, and another function may unwittingly intercept it. For example,

```
(define (foo)
  (catch 'foo (* 2 (bar))) )
(define (bar)
  (+ 1 (throw 'foo 5)) )
(foo)   → 5
```

Independently of the extent of the escape, **block** limits its reference to its body whereas **catch** authorizes that as long as it lives. As a consequence, it is possible to use the escape bound to **foo** all the time that **(* 2 (bar))** is being evaluated. In the preceding example, you cannot replace "**catch 'foo**" by "**block foo**" and expect the same results. An even more dangerous collision would be this:

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

Let's assume that we heard, next to the coffee-machine, that the function **better-map** is much better than **map**; and let's also suppose that we'll risk using it to test whether the result of **(hack-and-return-list)** is really a list made up of pairs; and furthermore, we'll assume that we don't know the definition of **better-map**, which happens to be this:

```
(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) )
```

In fact, the function **better-map** is quite interesting because it halts even if the list in the second argument is cyclic. Yet if **(hack-and-return-list)** returns the infinite list **#1=((foo . hack) . #1#)**[6] then **better-map** would try to escape. If that fact is not specified in its user interface, there might be a name conflict and a collision of escapes comparable to a twenty-car pile-up at rush hour. Here again we could change the names to limit such conflicts; doing so is simple with **catch** because it allows its label to be any possible Lisp value; in particular, the value could be a dotted pair that we construct ourselves. We could rewrite it more certainly this way then:

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

It is possible to simulate **block** by **catch** but there is no gain in efficiency in doing so. It suffices to convert the lexical escapes into dynamic ones, like this:

```
(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)
```

6. Here, we've used COMMON LISP notation for cyclic data. We could also have built such a list by **(let ((p (list (cons 'foo hack)))) (set-cdr! p p) p)**.

```
(throw label value) ) ) )
```

The macro **block** creates a unique label and binds it lexically to the variable of the same name. Doing so insures that only the **return-from**(s) present in its body will see that label. Of course, by doing that, we pollute the variable space with the name **label**. To offset that fault, we can try to use a name that doesn't conflict or even a name created by **gensym**, but once we do that, we have to make arrangements for the same name to appear in both **catch** and **throw**.

3.1.5 Escapes with Indefinite Extent

As part of its massive overhaul of Lisp around 1975, Scheme proposed that continuations caught by **catch** or **block** should have an indefinite extent. This property gave them new and very interesting characteristics. Later, in an attempt to reduce the number of special forms in Scheme, there was an effort to capture continuations by means of a function as well as to represent a continuation as a function, that is, as a first class citizen. In [Lan65], Landin suggested the operator J to capture continuations, and the function **call/cc** in Scheme is its direct descendant.

We'll try to explain its syntax as simply as possible at this first encounter. First of all, it involves the capture of a continuation, so we need a form that captures the continuation of its caller, k, like this:

$_k$(...)

Now since we want it to be a function, we'll name it **call/cc**, like this:

$_k$(call/cc ...)

Once we have captured k, we have to furnish it to the user, but how do we do that? We really can't return it as the value of the form **call/cc** because then we would render k to k. If the user is waiting for this value in order to use it in a calculation, it's so that he or she can turn this calculation into a unary function[7] since it is only an object that is waiting for a value to be used in a calculation. Accordingly, we could furnish this function as an argument to **call/cc**, like this:

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

In that way, the continuation k is turned into a first class value, and the function-argument of **call/cc** is invoked on it. The last choice due to Scheme is to not create a new type of object and thus to represent continuations by unary functions, indistinguishable from functions created by **lambda**. The continuation k is thus *reified*, that is, turned into an object that becomes the value of **k**, and it then suffices to invoke the function **k** to transmit its argument to the caller of **call/cc**, like this:

$_k$(call/cc (lambda (k) (+ 1 (k 2)))) → 2

Another solution would be to create another type of object, namely, continuations themselves. This type would be distinct from functions, and would necessitate a special invoker, namely, **continue**. The preceding example would then be rewritten like this:

$_k$(call/cc (lambda (k) (+ 1 (continue k 2)))) → 2

7. Warning to Scheme users: it is sufficient that the argument of call/cc should accept being called with only one argument. Then we could also write (call/cc list).

It is still possible to transform a continuation into a function by enclosing it in an abstraction (**lambda (v) (continue** *k* **v)**). Some people like calls to continuations that are syntactically eye-opening because they make alteration of the thread of control more evident.

The ultimate difficulty with **call/cc** is that its real name is **call-with-current-continuation**, a fact that does not improve the situation, so let's rewrite our example to use **call/cc**, like this:

```
(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) ) ) )
```

The call continuation of the function **find-symbol** is captured by **call/cc**, reified as a unary function bound to the variable **exit**. When the symbol is found, the escape is triggered by a call to the function **exit** (which never returns).

The indefinite extent of continuations is not obvious in that example since the continuation is used only during the invocation of the argument of **call/cc**, that is, during its dynamic extent. The following program illustrates indefinite extent.

```
(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)) ) )
```

The continuation reified in **c** and stored in **k** corresponds to this:

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

r→ 1
k≡ *k*
n

This continuation *k* appears as the value of the variable **k** that it encloses. Recursion, as we know, always implies something cyclic somewhere, and in this case, it is assured by the call to **k**, re-invoked until **n** reaches the threshold that we want. That whole effort, of course, computes the factorial.

In that example, you see that the continuation *k* is used outside it dynamic extent, which is equal to the time during which the body of the argument of **call/cc** is being evaluated. On that basis, you can imagine many other variations, for example, to submit the identity to **call/cc**, like this:

```
(define (fact n)
```

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

Now it's necessary for the recursive call to be the auto-application `(k k)` because, at every call of the continuation, the variable `k` is bound again to the argument provided to it. The continuation can be represented by this:

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

r→ 1
n

When we confer an indefinite extent on continuations, we make their implementation more problematic and generally more expensive. (Nevertheless, see [CHO88, HDB90, Mat92].) Why? Because we must abandon the model of evaluation as a stack and adopt a tree instead. In effect, when continuations have a dynamic extent, we can qualify them as escapes since their only goal is to escape from a computation. Escaping is the same as getting away from any remaining computations in order to impose a final value on a form that is still being evaluated. A computation begins when the evaluation of a form starts, and it seems simple enough to determine when it ends. In the presence of continuations with a dynamic extent, a computation always has a detectable end.

However, with continuations of indefinite extent, things are not so simple. Consider, for example, the expression `(call/cc ...)` in the preceding factorial. That expression returns a result many times. That fact (that a form can return values many[8] times) implies that the ultimate boundary of its computation is not necessarily the moment when it returns a result.

The function `call/cc` is very powerful and makes it possible in some ways to handle time. Escaping is like speeding up time until we provoke an event that we can foresee but that we foresee a long way off. Once it has escaped, a continuation with indefinite extent makes it possible to come back to that state. It's not just a means of convoluted looping because the values of the variables will generally have changed in the meantime, so coming back to a point already seen in the program forces the computation to take a new path.

We might compare `call/cc` to the "harmful" `goto` instruction of imperative languages. However, `call/cc` is more restricted since it is only possible to jump to places where we once passed through (after capture of that continuation) but not to places where we never went.

In the beginning, the function `call/cc` is sufficiently subtle to handle because the continuation is unary like the argument of `call/cc`. The following rewrite rule shows the effect of `call/cc` from another angle.

$_k$`(call/cc` ϕ`)` $\rightarrow$ $_k(\phi\ k)$

8. Returning values multiple times is not the same as returning multiple values, as in COMMON LISP.

In that rule, k represents the call continuation of `call/cc` and ϕ is some unary function. `call/cc` proceeds to reify the continuation k, an implementation entity, turning it into a value in the language that can appear as the second term of an application. Notice that the current continuation when ϕ is invoked is still k. Thus we are not compelled to use the continuation provided as an argument, and we implicitly depend on the same continuation, as you see in the following extract:

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

Some people might regret this fact, and like a strict authority figure, they might impose the rule that when we take the continuation, we remove it then from the underlying evaluator. In consequence, we would have to use that continuation to return a value, like this:

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

In that hypothetical world, `(k 1615)` would be evaluated with a continuation something like a black hole, say, $(\lambda u.\bullet)$. That continuation would absorb the value which would be rendered to it without further dissemination of information. Then there would be no posterior computation; your keyboard would dissolve into thin air; you would no longer exist, and neither would this book. In such a world, you certainly must not forget to invoke your continuation!

3.1.6 Protection

We need to look at one last effect linked to continuations. It has to do with the special form `unwind-protect`. That name comes from the period when the implementation of a trait determined its name[9] and indicated how to use it. Here's the syntax of the special form `unwind-protect`.

`(unwind-protect` *form*
cleanup-forms `)`

The term *form* is evaluated first; its value becomes the value of the entire form `unwind-protect`. Once this value has been obtained, the *cleanup-forms* are evaluated for their effect before this value is finally returned. The effect is almost that of `prog1` in COMMON LISP or of `begin0` in some versions of Scheme, that is, evaluating a series of forms (like `progn` or `begin`) but returning the value of the first of those forms. The special form `unwind-protect` guarantees that the cleanup forms will always be evaluated regardless of the way that computation of *form* occurs, even if it is by an escape. Thus:

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 ;and prints 2
```

9. The names `car` and `cdr` come from that same period; they are closely connected to the implementation of Lisp 1.5. They are acronyms for "contents of the address register" and "contents of the decrement register."

That form is interesting in the case of a subsystem where we want to insure that a certain property will be restored, regardless of the outcome in the subsystem. The conventional example is that of handling a file: we want to be sure that it will be closed even if an error occurs. Another more interesting example concerns a simulation of `catch` by `block`. Using `block` simultaneously would de-synchronize the variable `*active-catchers*`. We could correct that fault by means of a judicious, well placed `unwind-protect`, like this:

```
(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) ) ) ) ) )
```

Whatever the outcome of the computation carried out in the body, the list of active `catch` forms will always be up to date at the end of the computation. The `block` forms can thus be used freely along with the `catch` forms. The simulation is still not perfect because the variable `*active-catchers*` must be private to `catch` and `throw` (or, more accurately, private to their expansions). In the current state, we might alter it, either accidentally or intentionally, and thus destroy the order among these macros.

The form `unwind-protect` protects the computation of a form by insuring that a certain treatment will happen once the computation has been completed. Thus the form `unwind-protect` is inevitably tied to detecting when a computation is complete, and from there, to continuations with dynamic extent. For the reasons that we discussed earlier, `unwind-protect`[10] does not get along well with `call/cc` nor with continuations that have an indefinite extent.

As we have often emphasized before, the meaning of these constructions is not at all precise. Let's just consider a few questions about the values of the following forms:

```
(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 'no-associated-catch (return-from foo 2)) ) ) ) → ?
```

This kind of vagueness tends to make us want more careful and precise formality in these concepts. Nevertheless, we have to observe the equivocal status of cleanup forms and especially their continuations. We qualify the continuation as *floating*

10. An operational description of `unwind-protect` for Scheme, known as `dynamic-wind`, appears in [FWH92]; see also [Que93c].

because it cannot be known statically and may correspond to an escape that is already underway. Consider this, for example:

```
(block bar
   (unwind-protect (return-from bar 1)
       (block foo π2 ) ) )
```

The cleanup form captures the continuation corresponding to the lexical escape to **bar**. If the cleanup form returns a value, then this escape to **bar** will be taken back. If the cleanup form itself escapes, then its own continuation will be abandoned. (We'll discuss these phenomena later when we explain the implementation in greater detail.)

Other control forms also exist, and notably in COMMON LISP, where the old-fashioned form of **prog** metamorphoses into **tagbody** and **go**; they can be simulated by **labels** and **block**. **[see Ex. 3.3]** We'll come back to the fact that in a world of continuations with dynamic extent, **block**, **return-from**, and **unwind-protect** together provide the base of special forms that we usually find there. Similarly, in the world of Scheme, **call/cc** is all we need. It's clear that we cannot simulate **call/cc** directly with continuations with a dynamic extent. The inverse is feasible, but to do so, we have to tone down **call/cc**, which proves a little too powerful in that situation. The method for doing so will be clearer once we've explicated the interpreter with explicit continuations.

Protection and Dynamic Variables

Certain implementations of Scheme get their dynamic variables a little differently from the variations that we've looked at so far. They depend on the presence of the form **unwind-protect** or something similar. The idea involves a sort of lexical borrow and something of a theft. These dynamic variables are introduced by the form **fluid-let**, like this:

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

During the calculation of β, the variable **x** has the value α as its value; the preceding value of **x** is saved in a local variable, *tmp*, and it's restored at the end of the calculation. This form implies the existence of a lexical binding to borrow, usually a global binding, thus making the variable visible everywhere. If, in contrast, a local binding is borrowed, then (differing greatly from dynamic variables in COMMON LISP) this will be accessible in the body of the form **fluid-let** and only in it, with possibly pernicious effects on the co-owners of this variable. Moreover, the form **unwind-protect** and indefinite extent continuations don't get along together very well. Indeed, this form is even more subtle to use than dynamic variables in COMMON LISP.

3.2 Actors in a Computation

Now, from our current point of view, a computation is made up of three elements: an expression, an environment, and a continuation. While the immediate goal is to evaluate the expression in the environment, the long term goal is to return a value to the continuation.

Here we're going to define a new interpreter to highlight all the continuations needed at every level. Since continuations are usually represented as blocks or activation records in a stack or heap, we will use objects to represent them within the interpreter that we're undertaking.

3.2.1 A Brief Review about Objects

This section will not present an entire system of objects in all its glory; that task remains for Chapter 11. Here our goal is simply to present a set of three macros associated with a few naming rules. This set is merely the essence of a system of objects, and it's independent of any implementation of such a system. We chose to use objects in this chapter in order to suggest how to implement continuations. Objects with their fields conveniently evoke activation records that implement languages. The idea of inheritance, put to work here in a very elementary way, will let us factor out certain common effects and thus reduce the size of the interpreter that we're presenting.

We'll assume that you're familiar with the philosophy, terminology, and customary practice about objects, and we'll simply go over a few linguistic conventions that we'll be exploiting.

Objects are grouped into *classes*; objects in the same class respond to the same *methods*; messages are sent by means of *generic functions*, popularized by Common Loops [BKK+86], CLOS [BDG+88], and TEAOΣ[PNB93]. For our purposes, the interesting aspect of object-oriented programming is that we can organize various special forms or primitive functions for control around a kernel evaluator. The drawback of this style for us is that the dissemination of methods makes it harder to see the big picture.

Defining a Class

A class is defined by `define-class`, like this:

```
(define-class class superclass
      ( fields... ) )
```

This form defines a class with the name *class*, inheriting the fields and methods of its *superclass*, plus its own fields, indicated by *fields*. Once a class has been created, we can use a multitude of associated functions. The function known as `make-`*class* creates objects belonging to *class*; that function has as many arguments as the class has fields, and those arguments occur in the same order as the fields. The read-accessors for fields in these objects have a name prefixed by the name of the class, suffixed by the name of the field, separated by a hyphen. The write-accessors prefix the name of the read-accessor by `set-` and suffix it by an exclamation point.

Write-accessors do not have a specific return value. The predicate *class*? tests whether or not an object belongs to *class*.

The root of the inheritance hierarchy is the class `Object` having no fields.

The following definition:

```
(define-class continuation Object (k))     ;example
```

makes the following functions available:

```
(make-continuation k)          ;creator
(continuation-k c)             ;read-accessor
(set-continuation-k! c k)      ;write-accessor
(continuation? o)              ;membership predicate
```

Defining a Generic Function

Here's how we define a generic function:

```
(define-generic (function variables)
    [ default-treatment ... ] )
```

That form defines a generic function named *function*; *default-treatment* specifies what it does when no other appropriate method can be found. The list of variables is a normal list of variables except that it specifies as well the variable that serves as the *discriminator*; the discriminator is enclosed in parentheses.

```
(define-generic (invoke (f) v* r k)
  (wrong "not a function" f r k) )
```

That form defines the generic function **invoke**, which could be enriched with methods eventually. The function has four variables; the first is the discriminator: `f`. If there is a call to **invoke** and no appropriate method is defined for the class of the discriminator, then the default treatment, **wrong**, will be invoked.

Defining a Method

We use **define-method** to stuff a generic function with specific methods.

```
(define-method (function variables)
     treatment ... )
```

Just like the form **define-generic**, this form uses the list of variables to specify the class for which the method is defined. This class appears with the discriminator between parentheses. For example, the following form defines a method for the class **primitive**:

```
(define-method (invoke (ff primitive) vv* rr kk)
  ((primitive-address ff) vv* rr kk) )
```

That completes the set of characteristics that we'll be using. Chapter 11 will show you an implementation of this system of objects, but regardless of the implementation, for now, we'll only use the most widely known characteristics of objects and those least subject to hazards.

3.2.2 The Interpreter for Continuations

Now the function **evaluate** has three arguments: the expression, the environment, and the continuation. The function **evaluate** begins by syntactically analyzing the expression to determine appropriate treatment for it. Each of these treatments is individualized in a particular function. Before we get into the new interpreter, we should indicate the rules that we'll follow in naming our variables. The first convention is that a variable suffixed by a star represents a list of whatever the same variable without the star would represent.

e, **et**, **ec**, **ef** ...	expression, form
r ...	environment
k, **kk** ...	continuation
v ...	value (integer, pair, closure, etc.)
f ...	function
n ...	identifier

Now here's the interpreter. It assumes that anything that is atomic and unknown to it as a variable must be an implicit quotation.

```
(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)) ) ) )
```

That interpreter is actually built from three functions: **evaluate**, **invoke**, and **resume**. Only those last two are generic and know how to invoke applicable objects or handle continuations, whatever their nature. The entire interpreter will be little more than a series of hand-offs among these three functions. Nevertheless, two other generic functions will prove useful to determine or to modify the value of a variable; they are **lookup** and **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) )
```

All the entities that we'll manipulate will derive from three virtual classes:

```
(define-class value Object ())
(define-class environment Object ())
(define-class continuation Object (k))
```

The classes of values manipulated by the language that we are defining will inherit from **value**; the environments inherit from **environment**; and of course, the continuations inherit from **continuation**.

3.2.3 Quoting

The special form for quoting is still the simplest and consists of rendering the quoted term to the current continuation, like this:

```
(define (evaluate-quote v r k)
  (resume k v) )
```

3.2.4 Alternatives

An alternative brings two continuations into play: the current one and the one that consists of waiting for the value of the condition in order to determine which branch of the alternative to choose. To represent that continuation, we will define an appropriate class. When the condition of an alternative is evaluated, the alternative will choose between the true and false branch; consequently, those two branches must be stored along with the environment needed for their evaluation. The result of one or the other of those branches must be returned to the original continuation of the alternative, which must then be stored as well. Thus we get this:

```
(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) ) )
```

Then the alternative decides to evaluate its condition **ec** in the current environment **r**, but with a new continuation, made from all the ingredients needed for the computation. Once the computation of the condition has been completed by a call to **resume**, the computation of the alternative will get underway again and ask for the value of the condition **v** to determine which branch to choose, but in any case, it will evaluate it in the environment that was saved and with the initial continuation of the alternative.[11]

3.2.5 Sequence

A sequence also calls two continuations into play, like this:

```
(define-class begin-cont continuation (e* r))
(define (evaluate-begin e* r k)
```

11. Those who are implementation buffs should note that **make-if-cont** can be seen as a form pushing **et** and then **ef** and finally **r** onto the execution stack, the lower part of which is represented by **k**. Reciprocally, **(if-cont-et k)** and the others pop those same values.

```
  (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) ) )
```

The cases of `(begin)` and `(begin` π`)` are simple. When the form `begin` involves several terms, the first one must be evaluated by providing it a new continuation, `(make-begin-cont k e* r)`. When that new continuation receives a value by `resume`, it will trigger the method for `begin-cont`. That continuation will discard the value returned, `v`, and will restart the computation of the other forms present in `begin`.[12]

3.2.6 Variable Environment

The values of variables are recorded in an environment. That, too, will be represented as an object, like this:

```
(define-class null-env environment ())
(define-class full-env environment (others name))
(define-class variable-env full-env (value))
```

Two kinds of environments are needed: an empty environment to initialize computations, and instances of `variable-env` corresponding to non-empty environments. Those non-empty environments store a binding, that is, a name and a value; the rest of the environment is linked via the field `others`. Even though this way of organizing things uses objects, it is functionally similar to the A-list except that it consumes only an object of three fields rather than two dotted pairs.

Thus to search for the value of a variable, we do this:

```
(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) ) )
```

The generic function `lookup` scrutinizes the environment until it finds the right binding. The value determined that way is sent to the continuation by means of the generic function `resume`.

12. As an attentive reader, you will have noticed the form `(cdr (begin-cont-e* k))` presented in the method `resume`. Equivalently, we could have directly built it in `evaluate-begin`, like this: `(make-begin-cont k (cdr e*) r)`. The reason is that in case of error, analyzing the continuation will make it possible to know which expression was underway.

Modifying a variable entails the same process:

```
(define-class set!-cont continuation (n r))
(define (evaluate-set! n e r k)
  (evaluate e r (make-set!-cont k n r)) )
(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) ) )
```

It's necessary to introduce a particular continuation because the evaluation of an assignment is carried out in two phases: computing the value to assign and then modifying the variable involved. The class **set-cont!** implements this new kind of continuation; the adapted method **resume** merely calls the environment in order to update it.

3.2.7 Functions

Creating a function is a simple process, left to **make-function**.

```
(define-class function value (variables body env))
(define (evaluate-lambda n* e* r k)
  (resume k (make-function n* e* r)) )
```

What's a little more complicated is how to invoke functions. Notice the implicit **progn** or **begin** in the function bodies.

```
(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) ) )
```

It might seem unusual to you that the functions take the current environment **r** as an argument, even though they don't apparently use it. We've left it there for two reasons. One, there is often a register dedicated to the environment in implementations, and like any register, it's available. Two, certain functions (we'll see more of them later when we discuss reflexivity) can influence the current lexical environment, such as, for example, debugging functions.

The following function extends an environment. There is no need to test whether there are enough values or names because the functions have already verified that.

```
(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")) ) )
```

All that remains is to indicate how to define a functional application. In doing so, we should keep in mind that functions are invoked on arguments organized into a list.

```
(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 (evaluate-arguments e* r k)
  (if (pair? e*)
    (evaluate (car e*) r (make-argument-cont k e* r))
    (resume k no-more-arguments) ) )
(define 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) ) )
```

The technique is a little disconcerting at first glance. Evaluation takes place from left to right; the function term is thus evaluated first with a continuation of the class **evfun-cont**. When that continuation takes control, it proceeds to the evaluation of the arguments, leaving a continuation which will apply the function to them, once they have all been computed. During the evaluation of the arguments, continuations of the class **gather-cont** are left; their role is to gather the arguments into a list.

Let's take an example to see the various continuations that appear during the computation of **(cons foo bar)**. We'll assume that **foo** has the value 33 and **bar**, −77. To illustrate this computation, we'll draw the continuations as horizontal

stacks; k will be the continuation; r will be the current environment. We'll use *cons* to denote the global value of `cons`, that is, the allocation function for dotted pairs.

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 Initializing the Interpreter

Before we plunge into the arcane mysteries of control forms, let's sketch a few details about how to authorize execution of this interpreter. This section greatly resembles Section 1.7. [see p. 25] We first have to deck out the interpreter with a few well chosen variables, and to do so, we'll define some macros to enrich the global environment.

```
(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)
```

The primitives created have to be able to be invoked, like any other user-function, by `invoke`. These primitives each have two fields. The first makes de-

bugging easier because it indicates the original name of the primitive. Of course, that's only a hint because you can bind the primitive to other global names if you want.[13] The second field in any of those primitives contains the "address" of the primitive, that is, something executable by the underlying machine. A primitive is thus triggered like this:

```
(define-method (invoke (f primitive) v* r k)
  ((primitive-address f) v* r k) )
```

Then to start the interpreter, and begin to enjoy its facilities, we will define an initial continuation in a way similar to **null-env**. That initial continuation will print the results that we provide it.

```
(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) )
```

Notice that the entire interpreter could easily be written in a real object-language, like Smalltalk [GR83], so we could take advantage of its famous browser and debugger. The only thing left to do is to add whatever is needed to open a lot of little windows everywhere.

3.4 Implementing Control Forms

Let's begin with the most powerful control form, **call/cc**. Paradoxically, it is the simplest to implement if we measure simplicity by the number of lines. The style of programming we've been following—by objects—and the fact that we've made continuations explicit will almost make this implementation a trivial task.

3.4.1 Implementation of call/cc

The function **call/cc** takes the current continuation **k**, transforms it into an object that we can submit to **invoke**, and then applies the first argument, a unary function, to it. The lines that follow here express that idea equally strongly.

```
(definitial call/cc
  (make-primitive
   'call/cc
   (lambda (v* r k)
     (if (= 1 (length v*))
         (invoke (car v*) (list k) r k)
```

13. This same hint makes it possible in many systems to get the following effect: **(begin (set! foo car)(set! car 3) foo)** has a result that is printed as **#<car>** where the name of the primitive follows the functional object and not the binding through which it was found.

```
        (wrong "Incorrect arity" 'call/cc v*) ) ) ) )
```

Even though there are few lines there, we should explain them a little. Although it is a function, `call/cc` is defined by `definitial` because it needs access to its continuation so badly. The variable `call/cc` (now here we are in a Lisp_1) is thus bound to an object of the class `primitive`. The call protocol for these objects associates them with an "address" represented in the defining Lisp by a function with the signature `(lambda (v* r k) ... )`. Eventually, the first argument is applied to the continuation. That continuation has been delivered just as it is. Since the continuation might possibly be submitted to `invoke`, we remind ourselves that way to confer the *ad hoc* method on `invoke`.

```
(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 Implementation of `catch`

The form `catch` is interesting to implement because the mechanisms that it brings into play are quite different from those in `block`, which we will get to later. As usual, we'll add the necessary clauses to `evaluate` to analyze the forms `catch` and `throw`, like this:

```
...
((catch) (evaluate-catch (cadr e) (cddr e) r k))
((throw) (evaluate-throw (cadr e) (caddr e) r k)) ...
```

Here we've taken the option of making `throw` a special form, not a function taking a thunk. We did so to simulate COMMON LISP. As a special form, `catch` is defined like this:

```
(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) ) )
```

Now it is apparent that `catch` evaluates its first argument, binds that argument to its continuation by creating a tagged block, and then goes on with its work of sequentially evaluating its body. When a value is returned to the tagged block, that block is removed and simply transmits the value. The form `throw` will make better use of that tagged block.

```
(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-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)) ;comparator
      (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) )
```

The form **throw** first evaluates its first argument and then checks whether a continuation tagged that way exists. If not, it raises an error; otherwise, the value to transmit is then computed, and it will be transmitted to the continuation that's already been located. That continuation can not have changed. The evaluation of the value to transmit has a continuation that's a little special: an instance of **throwing-cont**. The reason: if by accident an error or control effect occurs in the computation of this value, the right context will be the current context, not the one that would have been adopted if everything had gone well. Thus we can write this:

```
(catch 2
  (* 7 (catch 1
         (* 3 (catch 2
                (throw 1 (throw 2 5)) )) )) )
```

The result is `(* 7 3 5)` and not 5. This definition of **throw** makes it possible to detect errors that could not have been caught if **throw** were a function.

```
(catch 2 (* 7 (throw 1 (throw 2 3))))
```

That form, for example, does not lead to 3 but to the error message, **"No associated catch"** since there is no **catch** form with tag 1.

3.4.3 Implementation of `block`

There are two problems to resolve in implementing lexical escapes. The first is to confer a dynamic extent on continuations. The second problem is to give lexical scope to the tags on lexical escapes. To do that, we'll use the lexical environment to confer the right scope. We'll define a new kind of environment to associate tags with continuations.

We'll add a clause to **evaluate**—a clause recognizing special **block** forms—and we'll define everything that follows.

```
(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) )
```

Now everything is in place for `return-from`, so we will add a clause to analyze `return-from` forms inside `evaluate`.

```
...
((block)       (evaluate-block (cadr e) (cddr e) r k))
((return-from) (evaluate-return-from (cadr e) (caddr e) r k)) ...
```

And we will also define this:

```
(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-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 (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) )
```

Once we've got the value to transmit, `block-lookup` searches for the continuation associated with the tag of the `return-from` in the lexical environment. If `block-lookup` finds it, then we verify whether the associated continuation is still valid by looking for it in the current continuation by means of the new function, `unwind`.

The search for an ad hoc block in the lexical environment is carried out by the generic function, **block-lookup**. We've defined the necessary methods for it so that it can skip environments for variables that don't really interest it in order to look only at instances of **block-cont**. Reciprocally, we've extended the generic function, **lookup** so that it ignores instances of **block-cont**. These methods have been defined for the virtual class **full-env** so that they can be shared in case other classes of environments are created.

The generic function **unwind** tries to transmit a value to a certain continuation which must still be alive, that is, it can be found in the the current continuation.

3.4.4 Implementation of unwind-protect

The form **unwind-protect** is the most complicated to handle; it implies modifications in the preceding definitions of the forms **catch** and **block** because they must be adapted to the presence of **unwind-protect**. It's a good example of a feature whose introduction alters the definition of everything that exists up to this point. However, not making use of **unwind-protect** implies a certain cost, too. The lines that follow here define **unwind-protect**, apparently only slightly different from **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)) )
```

Now, as we said, we have to modify **catch** and **block** to take into acount programmed cleanups when an **unwind-protect** form is breached by an escape. For **catch**, we'll modify the definition of **throwing-cont**, like this:

```
(define-method (resume (k throwing-cont) v)            ;** Modified **
  (unwind (throwing-cont-k k) v (throwing-cont-cont k)) )
(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) ) )
```

To transmit the escape value, we have to run back through the continuation a second time until we find the **target** continuation. During that rerun, if any **unwind-protect**s are breached, the associated cleanup forms will be evaluated. The continuation of these cleanup forms is from the class **unwind-cont** so it is possible to capture any of them. They correspond to pursuing the examination of the continuation—the floating continuations that we mentioned earlier on page 86.

With respect to **block**, the modification is the same kind but involves only **block-lookup**.

```
(define-method (block-lookup (r block-env) n k v) ; ** Modified **
  (if (eq? n (block-env-name r))
      (unwind k v (block-env-cont r))
      (block-lookup (block-env-others r) n k v) ) )
```

We look for the continuation of the associated **block** in the lexical environment, and then we unwind the continuation as far as this target block.

You might think that in the presence of **unwind-protect**, the form **block** is no faster than **catch** since they both share the heavy cost of **unwind**. Actually, since **unwind-protect** is a special form and thus its use can't be hidden, there are savings for all the **return-from**s that are not separated from their associated **block** by any **unwind-protect** or **lambda**.

Curiously enough, COMMON LISP (CLtL2 [Ste90]) introduced a restriction on escapes in cleanup forms. Those cleanup forms can't go *less far* than the escape currently underway. The intention of the restriction was to prevent any program being stuck in a situation where no escape could pull it out. **[see Ex. 3.9]** Consequently, the following program produces an error because the cleanup form wants to do less than the escape targeted toward 1.

COMMON LISP

```
(catch 1
  (catch 2
    (unwind-protect (throw 1 'foo)
      (throw 2 'bar) ) ) )          → error!
```

3.5 Comparing call/cc to catch

Thanks to objects, continuations look like linked lists of blocks. Some of these blocks are accessible from the lexical environment; others can be found only by running through the continuation, block by block. Still others give rise to various treatments when they are breached.

In a language such as Lisp, since it is blessed with continuations that have a dynamic extent, the idea of a stack is synonymous with the idea of continuation. When we write **(evaluate ec r (make-if-cont k et ef r))** we signify that we are pushing another block onto the stack indicating what to do on the return of the value of the condition of an alternative. Reciprocally, the expression **(evaluate-begin (cdr (begin-cont-e* k)) (begin-cont-r k) (begin-cont-k k))** says explicitly that the current block has been abandoned, popped in favor of the block just below: **(begin-cont-k k)**. We could verify that in such a language, no data structure keeps obsolete parts of continuations, that is, those parts that have disappeared from the stack. For that reason, when we leave a

block, the associated continuation (possibly captured elsewhere) is invalidated. In terms of implementation, continuations can be kept in a stack or even in several synchronized stacks and compiled into C primitives: `setjmp/longjmp`. [see p. 402]

In the dialect EULISP[PE92], there is a special form named `let/cc` with the following syntax:

```
(let/cc variable forms... )                                    EULISP
```

In Dylan [App92b], we write the same thing like this:

```
(bind-exit (variable) forms... )                               Dylan
```

This special form binds the current continuation to the *variable* with a scope equal to the body of the form `let/cc` in EULISP or `bind-exit` in Dylan. That continuation is consequently a first class entity with a unary functional interface. However, its *useful* extent is dynamic, so its use is limited to the evaluation time of the body of the form binding `let/cc` or `bind-exit`. More precisely, the object value of *variable* has an indefinite extent but a limited interest for the duration of the dynamic extent. This characteristic is typical of EULISP and Dylan, but it does not show up at all in Scheme (where continuations have an indefinite extent) nor in COMMON LISP (where continuations are not first class objects). However, we can simulate that behavior by writing this:

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc variable . body)
     (block variable
       (let ((variable (lambda (x) (return-from variable x))))
         . body ) ) ) ) )
```

In the world of Scheme, continuations can no longer be put on the stack because they can be kept in external data structures. Thus we have to adopt another model: a hierarchic model, sometimes called a *cactus stack*. The most naive approach is to leave the stack and allocate blocks for continuations directly in the heap.

This technique makes allocations uniform, and it makes porting easier, according to [AS94]. However, it decreases the locality of references in memory, and it means that we have to maintain the links between blocks explicitly. ([MB93], nevertheless, proposes solutions to those problems.) Generally, implementers work very hard to keep as many things as possible on the stack, so in this case, the canonical implementation of `call/cc` copies the stack into the heap; the continuation is thus this very copy of the stack. Other techniques have been studied, of course, as in [CHO88, HDB90], for sharing copies, delaying copies, making partial copies, etc. Each one of them, however, introduces certain costs.

The forms `block` and `call/cc` are more similar than are `catch` and `call/cc`. They share the same lexical discipline; they are distinguished only by the extent of their continuation. There's a restricted variation of `call/cc` in certain dialects, as in [IM89]; the restriction is known as `call/ep` for *call with exit procedure*; it is quite apparent in `block/return-from` as well as in the preceding `let/cc`. The function `call/ep` has the same interface as `call/cc`, like this:

```
(call/ep (lambda (exit) ...))
```

The variable **exit** in the unary function (the argument of **call/ep**) is bound to the continuation of the form **call/ep** limited to its dynamic extent. The resemblance to **block** is quite clear here except that instead of using a disjoint name space (the lexical escape environment), we borrow the name space for variables. The main difference that **call/ep** brings in is that the continuation of an escape becomes first class and can thus be handled like any other first class object. However, if we have to use **block**, then we must explicitly construct that first-class value if we need it. To do so, we write **(lambda (x) (return-from** *label* **x))**. The expression is equally powerful, but all the calling sites (that is, the **return-from** forms) are known statically in a **block** form. That's not always the case during a call to **call/ep**: for example, **(call/ep foo)** doesn't say anything about the use of an escape except after more refined analysis, an analysis not local to **foo**. Consequently, the function **call/ep** complicates the work of the compiler as compared to what a special form like **block** requires.

When we compare **block** and **call/ep**, we thus see some differences. For one, an efficient execution must not create the argument closure of **call/ep** if it is an explicit **lambda** form. Thus at compilation, we must distinguish the case of **(call/ep (lambda ...))** since it can be compiled better. This particular case is comparable to the way a special form is handled since both of them lead to treatment that is distinct within the compiler. Functions are the favorite vehicles for adepts of Scheme whereas special forms correspond more nearly to a kind of declaration that makes life easier for the compiler. Both are often equally powerful, though their complexity is different for both the user and the implementer.

In summary, if you're looking for power at low volume, then **call/cc** is for you since in practice it lets you code every known control structure, namely, escape, coroutine, partial continuation, and so forth. If you need only "normal" things (and Lisp has already demonstrated that you can write many interesting applications without **call/cc**), then choose instead the forms of COMMON LISP where compilation is simple and the generated code is efficient.

3.6 Programming by Continuations

There's a style of programming based on continuations. It entails explicitly telling a computation where to send the result of that computation. Once the computation has been achieved, the executor applies the receiver to the result, rather than returning it as a normal value. What that boils down to is this: if we have the computation **(foo (bar))**, we modify the function **bar** into **new-bar** in order to take an argument; that argument will be exactly what we will call the continuation. In the present case, that's **foo**. The modified computation will thus appear as **(new-bar foo)**. Let's look at an example with our familiar friend, the factorial; let's assume that we want to compute $n(n!)$:

```
(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!)
```

The factorial thus takes a new argument `k`, the receiver of the final result. When the result is 1, it is simple to apply `k` to it. However, when the result is not immediate, we recall recursively as expected. Now the problem is to do two things at once: to transmit the receiver and to multiply the factorial of `n-1` by `n`. And furthermore, since we want multiplication to remain a commutative operation, we have to do these things in order! First, we'll multiply by `n` and then call the receiver. Since that receiver will be applied to 1 in the end, there's nothing left to do but compose the receiver and the supplementary handling and thus get `(lambda (r) (k (* n r)))`.

The advantage this new factorial offers us is that the same definition makes it possible to calculate many results, namely, the factorial `(fact` n `(lambda (x) x))`, the double factorial `(fact` n `(lambda (x) (* 2 x)))`, etc.

3.6.1 Multiple Values

Using continuations is also the key to multiple values. When a computation has to return multiple results, continuations become an interesting technique for doing so. In COMMON LISP, division (i.e., `truncate`) returns a multiple value composed of the divisor and the remainder. We could get a similar operator—one that we'll call `divide`—to take two numbers and a continuation, compute the quotient and the Euclidean remainder of those two numbers, and apply the continuation of the third argument to them. Then, to verify whether a division is correct, we could just do this:

```
(let* ((p (read)) (q (read)))
  (divide p q (lambda (quotient remainder)
                (= p (+ (* quotient q) remainder)) )) )
```

An example with more depth involves calculating Bezout numbers.[14] The Bezout identity stipulates that if p and q are relatively prime, then there must exist numbers u and v such that $up + vq = 1$. To compute Bezout numbers, we first have to compute the gcd (greatest common divisor) verifying as we do so that p and q are relatively prime.

```
(define (bezout n p k)     ;assume n > p
  (divide
   n p (lambda (q r)
         (if (= r 0)
             (if (= p 1)
                 (k 0 1)   ;since 0 × 1 − 1 × 0 = 1
                 (error "not relatively prime" n p) )
             (bezout
              p r (lambda (u v)
                    (k v (- u (* v q))) ) ) ) ) ) )
```

The `bezout` function uses `divide` to fill in `q` and `r` as the quotient and remainder of the division of `n` by `p`. If the numbers are really relatively prime, there's a trivial solution of 0 and 1. Otherwise, we keep increasing those values until we hit the initial `n` and `p`. We could verify the underlying mathematics; to do so, it suffices to know enough number theory to understand gcd or (proof by fire!) to verify that

14. Oof! I finally eventually succeed in publishing that function, first written in 1981!

```
(bezout 1991 1960 list) → (-569 578)
```

3.6.2 Tail Recursion

In the example of the factorial with continuations, a call to `fact` generally turned out to be nothing other than a call to `fact`. If we trace the computation of `(fact 3 list)`, but skip the obvious steps, we get this:

```
(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
≡ (k (* n 2))
              | n→ 3
              | k≡ list
→ (6)
```

When the call to `fact` is translated directly into a call to `fact`, the new call is carried out with the same continuation as the old call. This property is known as *tail recursion*—recursion because it is recursive and tail because it's the last thing remaining to do. Tail recursion is a special case of a tail call. A tail call occurs when a computation is resolved by another one without the necessity of going back to the computation that's been abandoned. A call in *tail position* is carried out by a *constant continuation.*

In the example of the `bezout` function, `bezout` calls the function `divide` in tail position. The function `divide` itself calls its continuation in tail position. That continuation recursively calls `bezout` in tail position again.

In the "classic" factorial, the recursive call to `fact` within `(* n (fact (- n 1)))` is said to be *wrapped* since the value of `(fact (- n 1))` is taken again in the current environment to be multiplied by `n`.

A tail call makes it possible to abandon the current environment completely when that environment is no longer necessary. That environment thus no longer deserves to be saved, and as a consequence, it won't be restored, and thus we gain considerable savings. These techniques have been studied in great detail by the French Lisp community, as in [Gre77, Cha80, SJ87], who have thus produced some of the fastest interpreters in the world; see also [Han90].

Tail recursion is a very desirable quality; the interpreter itself can make use of it quite happily. Optimizations linked to tail recursion are based on a simple point: on the definition of a sequence, of course. Up to now, we have written this:

```
(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) ) )
```

We could have also written it more simply like this:

```
(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-cont-e* k))))
   (if (pair? e*)
       (evaluate-begin e* (begin-cont-r k) (begin-cont-k k))
       (resume (begin-cont-k k) v) ) ) )
```

That first way of writing it is preferable because when we evaluate the last term of a sequence, with the first way, we don't have to build the continuation `(make-begin-cont k e* r)` to find out finally that it's equivalent to `k`, and building that continuation, of course, is costly in time and memory. In keeping with our usual operating principle of good economy in not keeping around useless objects unduly (notably the environment `r` in `(make-begin-cont k e* r)`), it's better to get rid of that superfluous continuation as soon as possible. This case is very important because every sequence has a last term!

The same effect can be applied to the evaluation of arguments, so we will write the following for all the same reasons we gave before about sequences:

```
(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 no-more-argument-cont) v)
  (resume (no-more-argument-cont-k k) (list v)) )
```

Here, we've written a new kind of continuation that lets us make a list out of the value of the last term of a list of arguments without having to keep the environment `r`. Friedman and Wise in [Wan80b] are credited with discovering this effect.

3.7 Partial Continuations

Among the questions that continuations raise, there's one about the following continuation: during an escape, what becomes of the part that disappears? In other words, the portion of the continuation (or "execution stack") which is located between the place from which we escape and the place where we jump represents a continuation slice. This slice takes an input value, and since it has an end, it also provides a result on exit. Thus it is equivalent to a unary function. Many researchers, such as [FFDM87, FF87, Fel88, DF90, HD90, QS91, MQ94], have elaborated control forms to reify these slices, or *partial continuations*, as they are known.

Consider for a moment the following simplified actions:

```
(+ 1 (call/cc (lambda (k) (set! foo k) 2)))   → 3
(foo 3)                                       → 4
```

In conformity with what we've already said, the continuation `k`, assigned to `foo`, is $\lambda u.1 + u$. But then what is the value of `(foo (foo 4))`?

```
(foo (foo 4))                                 → 5
```

The result is 5, not 6 as composing the continuations might lead you to think. In effect, calling a continuation corresponds to abandoning a computation that is underway and thus at most only one call can be carried out. Thus the call inside `foo` forces the computation of the continuation $\lambda u.1+u$ for 4; its mission is to return the value that's gotten as the final value of the entire computation. Consequently, we never get back from a continuation! More specifically, the continuation `k` has to wait for a value, add 1 to it, and return the result as the *final and definitive* result.

The same example is even more vivid with contexts. The continuation of the example was `(+ 1 [])`. Since we are in a call by value, `(foo (foo 4))` corresponds to eliminating the context `(foo [])` around `(foo 4)` so that we rewrite it as `(+ 1 4)`, and its final value is 5.

Partial continuations represent the followup of computations that remain to be carried out up to a certain well identified point. In [FWFD88, DF90, HD90, QS91], there are proposals about how to make continuations partial and thus composable. Let's assume that the continuation bound to `foo` is now `[(+ 1 [])]`, where the external square brackets indicate that it must return a value. Then `(foo (foo 4))` is really `(foo [(+ 1 [4])])` since `(+ 1 5)` eventually results in 6. The captured continuation `[(+ 1 [])]` does not define all the rest of the computation but only what remains up to but not including the return to the toplevel loop. Thus the continuation has an end, and it's thus a function and composable.

Another way of looking at this effect is to take the example again that we saw earlier—the one that has the side effect on the global variable `foo`—as well as an interaction with the toplevel loop. Let's condense the two expressions into one and write this:

```
(begin (+ 1 (call/cc (lambda (k) (set! foo k) 2)))
       (foo 3) )
```

Kaboom! That loops indefinitely because `foo` is now bound to the context `(begin (+ 1 []) (foo 3))` which calls itself recursively. From this experiment, we can conclude that gathering together the forms that we've just created is not a neutral

activity with respect to continuations nor with respect to the toplevel loop. If we really want to simulate the effect of the toplevel loop better, then we could write this:

```
(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))) )
```

Every time that `toplevel` gets a form to evaluate, we store a continuation in the variable `sequel`—the continuation leading to the next form to evaluate. Every continuation gotten during evaluation is thus by this fact limited to the current form. As you've observed, when `call/cc` is used outside its dynamic extent, there must be either one of two things: either a side effect, or an analysis of the received value to avoid looping.

Partial continuations specify up to what point to take a continuation. That specification insures that we don't go too far and that we get interesting effects. We can thus rewrite the preceding example to redefine `call/cc` so that it captures continuations only up to `toplevel` and no further. The idea of escape is also necessary here to eliminate a slice of the continuation. Even so, for us, partial continuations are still not really attractive because we're not aware of any programs using them in a truly profitable way and yet being any simpler than if rewritten more directly. In contrast, what's essential is that all the operators suggested for partial continuations can be simulated in Scheme with `call/cc` and assignments.

3.8 Conclusions

Continuations are omnipresent. If you understand them, then you've simultaneously gained a new programming style, mastered the intricacies of control forms, and learned how to estimate the cost of using them. Continuations are closely bound to execution control because at any given moment, they dynamically represent the work that remains to do. For that reason, they are highly useful for handling exceptions.

The interpreter presented in this chapter is quite precise yet still locally readable. In the usual style of object programming, there are a great many small pieces of code that make understanding of the big picture more subtle and more challenging. This interpreter is quite modular and easily supports experiments with new linguistic features. It's not particularly fast because it uses up a great many objects, usually only to abandon them right away. Of course, one of the roles of a

compiler is to determine just which entities would be useful to build anyway.

3.9 Exercises

Exercise 3.1 : What is the value of `(call/cc call/cc)`? Does the evaluation order influence your answer?

Exercise 3.2 : What's the value of `((call/cc call/cc) (call/cc call/cc))`?

Exercise 3.3 : Write the pair **tagbody/go** with **block**, **catch**, **labels**. Remember the syntax of **tagbody**, as defined in COMMON LISP:

```
(tagbody
        expressions0...
  label1 expressions1...
                ...
  labeli expressionsi...
                ... )
```

All the expressions *expressions*$_i$ but only the expressions *expressions*$_i$ may contain unconditional branching forms (**go** *label*) or escapes (**return** *value*). If no **return** is encountered, then the final value of **tagbody** is **nil**.

Exercise 3.4 : You might have noticed that, during the invocation of functions, they verify their actual arity, that is, the number of arguments submitted to them. Modify the way functions are created to precompute their arity so that we can speed up this verification process. You only have to handle functions with fixed arity for this exercise.

Exercise 3.5 : Define the function **apply** so that it is appropriate for the interpreter in this chapter.

Exercise 3.6 : Extend the functions recognized by the interpreter in this chapter so that the interpreter accepts functions with variable arity.

Exercise 3.7 : Modify the way the interpreter is started so that it calls the function **evaluate** only once.

Exercise 3.8 : The way continuations are presented in Section 3.4.1 means that the code mixes continuations and values. Since instances of the class **continuation** may appear as values, we were obliged there to define a method of **invoke** for **continuation**. Redefine **call/cc** to create a new subclass of **value** corresponding to reified continuations.

Exercise 3.9 : In COMMON LISP, write a function `eternal-return` to take a thunk as its argument, to call it without stopping, and to make sure that no escape attempted by the thunk will succeed in getting out of the function `eternal-return`.

Exercise 3.10 : Consider the following function, due to Alan Bawden:

```
(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 ) )
```

If we assume that `box1` has, as its value, the result of `(make-box 33)`, then what is the value of the following expressions?

```
(box1 'get)
(begin (box1 'set 44) (box1 'get))
```

Exercise 3.11 : The function `evaluate` is the only one that is not generic. If we want to create a class of programs with as many subclasses as there are different syntactic forms, then the reader would no longer have recourse to an S-expression but to an object corresponding to a program. The function `evaluate` would thus be generic, and we could add new special forms easily and even incrementally. Redesign the interpreter to make that possible.

Exercise 3.12 : Define `throw` as a function instead of a special form.

Exercise 3.13 : Compare the execution speed of normal code and that translated into CPS.

Exercise 3.14 : Program `call/cc` by means of `the-current-continuation`. Assume that `the-current-continuation` is defined like this:

```
(define (the-current-continuation)
   (call/cc (lambda (k) k)) )
```

Recommended Reading

A good, non-trivial example of the use of continuations appears in [Wan80a]. You should also read [HFW84] about the simulation of messy control structures. In the historical perspective in [dR87], the rise of reflection in control forms is clearly stressed.

4
Assignment and Side Effects

In the previous chapters, with their spiraling build-up of repetition and variations, you may have felt like you were being subjected to the Lisp-equivalent of Ravel's *Bolero*. Even so, no doubt you noticed two motifs were missing: assignment and side effects. Some languages abhor both because of their nasty characteristics, but since Lisp dialects procure them, we really have to study them here. This chapter examines assignment in detail, along with other side effects that can be perpetrated. During these discussions, we'll necessarily digress to other topics, notably, equality and the semantics of quotations.

Coming from conventional algorithmic languages, assignment makes it more or less possible to modify the value associated with a variable. It induces a modification of the state of the program that must record, in one way or another, that such and such a variable has a value other than its preceding one. For those who have a taste for imperative languages, the meaning we could attribute to assignment seems simple enough. Nevertheless, this chapter will show that the presence of closures as well as the heritage of λ-calculus complicates the ideas of binding and variables.

The major problem in defining assignment (and side effects, too) is choosing a formalism independent of the traits that we want to define. As a consequence, neither assignment nor side effects can appear in the definition. Not that we have used them excessively before; the only side effects that appeared in our earlier interpreters were localized in the function `update!` (when it involved defining assignment, of course) and in the definition of "surgical tools" like `set-car!` (in the Lisp being defined) which was only an encapsulation of `set-car!` (in the defining Lisp).

4.1 Assignment

Assignment, as we mentioned, makes it possible to modify the value of a variable. We can, for example, program the search for the minimum and maximum of a binary tree of natural integers by maintaining two variables containing the largest and smallest values seen so far. We could write that this way:

```
(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) ) )
```

The function **min-max** is easy to understand and has the advantage that it uses only two dotted pairs to return the result we want. The algorithm is very much like one we could write in Pascal, and it's no less representative of the conventional use for variables. Notice, too, that the side effects it perpetrates on these local variables are completely invisible from outside the function **min-max** and thus do no damage to the general quality of the surrounding program. This is an example of a healthy side effect; it's clear and efficient, compared to purely functional versions of the same program. [**see Ex. 4.1**]

The assignment of an unshared local variable poses hardly any problems. For example, the following function enumerates the natural numbers, starting from zero. It's obvious there that asking for the value of the variable **n** immediately after assigning it is sure to return the new value that it's taken.

```
(define enumerate
  (let ((n -1))
    (lambda () (set! n (+ n 1))
               n ) ) )
```

Every call to **enumerate** returns a number. The function **enumerate** has an internal state, represented by the number **n**, which is modified progressively each time **enumerate** is called. Altering a closed variable is a problem in itself. λ-calculus is silent about this effect that exists only because we want to offer assignment within the language. To apply a function in mathematics, like in λ-calculus, we substitute for its variables the values or expressions that they take during the application. Assignment forces us to abandon this semantics, and it makes programs that use it lose their referential transparency.

If, for example, we replace all the occurrences of the variable **n** by its initial value in **enumerate**, then there would be no question of that function generating all the natural numbers, and **enumerate** would return the value **-1** eternally. Thus assignment forces us to abandon instantaneous substitution of variables by their values as a mode of computation. Substitution is now deferred in time, and it's carried out only when we want a specific value for a variable. Consequently, there are ways of interpolating assignments to modify substitutions that have not occurred yet.

Another example will highlight that problem more clearly. Consider this program:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (set! set-winner! (lambda (new-name) (set! name new-name)
                                       name ))
  (set-winner! "Me")
  (winner) )
```

Will the call to `(winner)` return `"Nemo"` or `"Me"`? In other words, are the modifications belonging to `set-winner!` perceived by `winner`?

Once more, λ-calculus is silent about this subject, and there's a good reason for its reticence: the idea of assignment is completely foreign to λ-calculus. Even so, we have said that the creation of a function should capture its definition environment; that is, the functions `winner` and `set-winner!` should store the fact that the variable `name` has the value `"Nemo"` at the time of their creation. It seems obvious that the form `(set-winner! "Me")` is going to return `"Me"` since the text of the function declares that we assign the variable `name` and then we return its value. The problem is to determine whether `(winner)` sees this new value, since it sees the same variable `name`.

A literal way of looking at the fact that a closure is defined in the environment where the free variables in its body have values that they had when it was created works in favor of isolation. In consequence, we could not distinguish the preceding program from the following one:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (winner) )
```

[Sam79] proposed that assignment should be visible only to the one who makes the assignment. In that case, the function `set-winner!` would return the modified value while `winner` always continued to return only `"Nemo"`. In that world, the idea of binding doesn't even exist. There is a connection between a variable and a value, but that connection is direct. The reference to a variable thus consults the current environment and returns the associated value. Assignment creates a new environment equivalent to the old one except that the assigned variable now has the assigned value; this new environment is provided as the current one to the continuation.

This way of looking at things recalls the traditional way of implementing closure in old dynamic Lisps. A closure was created explicitly by the special form `closure`; it took a list of variables to close as its first term, and as its second term, it took a function. For example, the form `(closure (x) (lambda (y) (+ x y)))` leads to `(lambda (y) (let ((x '`*value-of-*`x)) (+ x y)))`. There's capture of the value associated with the variable `x`, and that seems like it conforms to the philosophy inherited from λ-calculus.

However, that model does not support assignment very well. It also makes the idea of a shared variable problematic. Other solutions have also been suggested; [SJ87] recommends a form of `closure` that analyzes the body of the function to close so that it automatically extracts the free variables to capture. We could also modify assignments found in the body of the function to close in such a way as to fix the difficult problem of updating close variables, as suggested in [BCSJ86, SJ93].

Scheme tackles the problem differently by introducing the idea of ***binding*** and

achieves a number of interesting programming effects that way. The form `let` introduces a new binding between the variable **name** and the value **"Nemo"**. Functions created in the body of `let` capture, not the value of the variable, but its binding. The reference to the variable **name** is thus interpreted as a search for the binding associated with the variable **name** and then the extraction of the value associated with the binding that has already been discovered.

Assignment proceeds in the same way: assignment searches for the binding associated with the variable **name** and then alters the associated value inside this binding. The binding is thus a second class entity; a variable references a binding; a binding designates a value. Assigning a variable does not change the binding associated with it but modifies the contents of this binding to designate a new value.

4.1.1 Boxes

To flesh out the idea of binding, let's look again at A-lists. In the preceding interpreters, an A-list served us as the environment for variables, simply acting like a backbone to organize the set of variable-value pairs. A variable-value pair is represented by a dotted pair there. We search for that dotted pair when the variable is read- or write-referenced. That dotted pair (or, more precisely, its `cdr`) is modified by assignment. For this representation, then, we can identify the binding with this dotted pair. Other kinds of encoding are also possible; in fact, bindings can be represented by *boxes*. This transformation is significant because it lets us free ourselves entirely from assignments strictly in favor of side effects.

A box is created by the function **make-box**, conferring its initial value. A box can be read and written by the functions **box-ref** and **box-set!**. If we try a first draft of the function in message passing style, it would look like this:

```
(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) )
```

A way of implementing it without closure would use dotted pairs directly, like this:

```
(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) )
```

More briefly, we could simply use `define-class` and just write this:

```
(define-class box Object (content))
```

In those three ways (and you can see a fourth way in Exercise 3.10), we highlight the indeterminism of the value that **box-set!** returns. Every variable that submits to one or more assignments can be transcribed in a box which can be implemented conveniently. It is easy then to determine whether a variable is mutable; we do so by looking at the body of the form which binds it to see whether the variable is the object of an assignment. (One of the attractive features of lexical languages is that all the places where a local variable might be used are visible.)

We will specify how to box a variable by rewrite rules, where π will be transformed into $\overline{\pi}^v$ and v is the name of the variable to box.

$$
\begin{aligned}
\overline{x}^v &= \text{if } x = v \text{ then } (\texttt{box-ref } v) \text{ else } x \\
\overline{(\texttt{quote } \epsilon)}^v &= (\texttt{quote } \epsilon) \\
\overline{(\texttt{if } \pi_c\ \pi_t\ \pi_f)}^v &= (\texttt{if } \overline{\pi_c}^v\ \overline{\pi_t}^v\ \overline{\pi_f}^v) \\
\overline{(\texttt{begin } \pi_1 \ldots \pi_n)}^v &= (\texttt{begin } \overline{\pi_1}^v \ldots \overline{\pi_n}^v) \\
\overline{(\texttt{set! } x\ \pi)}^v &= \text{if } x = v \text{ then } (\texttt{box-set! } v\ \overline{\pi}^v) \\
&\quad \text{else } (\texttt{set! } x\ \overline{\pi}^v) \\
\overline{(\texttt{lambda } (\ldots x \ldots)\ \pi)}^v &= \text{if } v \in \{\ldots x \ldots\} \text{ then } (\texttt{lambda } (\ldots x \ldots)\ \pi) \\
&\quad \text{else } (\texttt{lambda } (\ldots x \ldots)\ \overline{\pi}^v) \\
\overline{(\pi_0\ \pi_1 \ldots \pi_n)}^v &= (\ \overline{\pi_0}^v\ \overline{\pi_1}^v \ldots \overline{\pi_n}^v)
\end{aligned}
$$

As usual, we must be careful in correctly handling local variables that have the same name as the variable that we want to rewrite in a box. By re-iterating the process on all the variables that are the object of an assignment that is all mutable variables, we completely suppress assignments in preference to boxes where the contents can be revised by side effects. We'll thus add the following rule:

```
(lambda (...x...) π)   ∧  (set! x ...) ∈ π
→  (lambda (...x...) (let ((x (make-box x))) π^x))
```

Let's look at the preceding example again and rewrite it in terms of boxes to get this:

```
(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) )
```

Using boxes in place of mutable variables (that is, assignable ones) is conventional and corresponds to "references" in dialects of ML. Boxes offer the advantage of (apparently) suppressing assignments and the problems connected with them. They also make it possible to avoid introducing special cases in the search for the value of a variable since no variables can be modified under those circumstances. Since they are pure side effects, they make us lose referential transparency, but they lend themselves to typing. On the other hand, boxes introduce two problems.

The first is that a binding, now represented by a box, becomes a first class object and can be manipulated as such. In other words, **box-ref** and **box-set!** are not the only operations that can be applied to boxes. Two people aware of the same box create an alias effect that can be used to implement modules.

The second problem is that the future of a box is usually indeterminate. Like a dotted pair that can be subjected to a disastrous **set-car!** by just about any-

body, the places where boxes are used are generally unknown. In contrast, lexical assignment has this going for it that all the sites where a binding can be altered are statically known. Compilations can take advantage of such knowledge when, for example, an assigned variable is neither closed nor shared; such is the case for the variables **min** and **max** in the function **min-max**.

Each assignable variable is associated with a place in memory (that is, its address) containing the value of the binding. That location in memory will be modified when the variable is assigned.

4.1.2 Assignment of Free Variables

Another problem involving assignment is the meaning to impute to it for a free variable. Consider this example:

```
(let ((passwd "timhukiTrolrk"))    ; That's a real password!
  (set! can-access? (lambda (pw) (string=? passwd (crypt pw)))) )
```

The variable **can-access?** is free in the body of **let**. Moreover, it is also assigned. When we follow the rules of Scheme, the variable **can-access?** must be global since it is not claimed locally. But the fact that the environment should be global does not necessarily signify that it contains the variable **can-access?**! We debated a similar topic earlier [see p. 54] where we saw several different possible solutions.

What can we do with a global variable, other than assign it? Like any other variable, we can reference it, close it, get its value, and in general define it before any other operation. The global environment itself is a name space, and we're going to see many different ways of producing it.

Universal Global Environment

The global environment can be defined as the place where all variables pre-exist. In fact, every time a new variable name appears, the implementation manages to make sure that the variable by that name is present in the global environment as though it had always been there, according to [Que95]. In that world, for every name, there exists one and only one variable with that name. The idea of defining a global variable makes no sense since all variables pre-exist there. Modifying a variable poses no problem since its existence cannot be in doubt. Consequently, we can reduce the operator **define** to a mere **set!** and, as a corollary, multiple definitions of the same variable are possible.

Only one problem crops up when we want to get the value of a variable which has not yet been assigned. That variable exists, but does not yet have a value. This type of error often goes under the name of an *unbound variable* message, even though in the strictest sense, the variable does indeed have a binding: its associated box. In other words, the variable exists but is uninitialized.

We can summarize the properties of this environment in the following chart.

Reference	x
Value	x
Modification	`(set!` x ...`)`
Extension	no
Definition	no, `define`≡`set!`

This definition environment is more interesting than it might appear at first glance. For one thing, it does not need many concepts because everything in it pre-exists. That makes it easy to write mutually recursive functions. In case of error, it lets us redefine global functions. That characteristic means that any reference to a global variable has to be handled with care since *(i)* it might not be initialized yet (but once initialized, it's permanent); *(ii)* it can change value—a fact that prohibits any hypothesis based on its current value. In particular, we must not even *inline* the primitive functions `car` and `cons`, but we can automatically recompile anything that depends on a hypothesis which has just been trashed.

To illustrate this environment, consider the following fragment, showing various properties.

```
g                          ;error: g uninitialized
(define (P m) (* m g))
(define g 10)
(define g 9.81)            ;≡ (set!  g 9.81)
(P 10)              → 98.1
(set! e 2.78)              ;definition of e
```

In summary, you can think of the global environment as one giant `let` form, defining all variables, something like this:

```
(let (... a aa  ... ab  ... ac ... )
  ... )
```

Frozen Global Environment

Now imagine that for every name there is at most one global variable by that name and that the set of defined names is immutable. This is the situation of a compiled, autonomous application without dynamically created code (that is, no calls to `eval`). That was also the situation of the preceding interpreters that did not authorize the creation of new global variables. They had to be created explicitly by the form `definitial` in the implementation language.

In such an environment, a global variable exists only after having been created by `define`. We can get or modify its value only if it has been defined beforehand. Yet, since there can be only one variable by any given name, we can reference such a variable even before it is defined. (That's the feature that allows mutual recursion.) However, we cannot define a global variable more than once. We'll summarize those properties in our familiar chart.

Reference	x
Value	x but x must exist
Modification	`(set!` x ...`)` but x must exist
Extension	`define` (only one time)
Definition	no

Now for this environment, let's look again at the preceding fragment to see where it leads this time.

```
g                          ;error: no variable g
(define (P m) (* m g))     ;forward reference to g
(define g 10)
(define g 9.81)            ;error: redefinition of g
(set! g 9.81)              ;modification of g
(P 10)            → 98.1
(set! e 2.78)              ;error: no variable e
```

This environment is beginning to suggest the idea of a program. A program is defined by a set of expressions $\pi_1 \ldots \pi_n$ that we can organize into a single expression built like this: we put the forms $\pi_1 \ldots \pi_n$ into a unique **let** form introducing all the free variables present in $\pi_1 \ldots \pi_n$ as non-initialized local variables; and then we modify all the **define** forms by changing them into the equivalent **set!** forms.

To clarify those ideas, here's a little application written in Scheme:

```
(define (crypt pw) ...)
(let ((passwd "timhukiTrolrk"))
  (set! can-access? (lambda (pw) (string=? passwd (crypt pw)))) )
(define (gatekeeper)
  (until (can-access? (read)) (gatekeeper)) )
```

That little application asks the user for a password and won't let the user through unless he or she supplies the right one. That computerized version of Cerberus is equivalent to this:

```
(let (crypt make-can-access? can-access? gatekeeper)
  (set! crypt (lambda (pw) ...))
  (set! make-can-access?
        (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
...
```

Global definitions are transformed into local definitions assigning the free variables of the application, re-organized and still non-initialized, in an all-encompassing **let**.[1] Of course, the usual functions, like **read**, **string=?**, **string-reverse**, or **string-append**, are visible. In this world, the global environment is finite and restricted to predefined variables (like **car** and **cons**) and to free variables of the program. However, it is not possible to assign a free variable not present in the global environment since that very environment was designed to avoid such a possibility. The only way to provoke that error would be to have a form of **eval** that allowed dynamic evaluation of code.

1. That definition of a program inadvertently makes multiple definitions of the same variable meaningful; for that reason, we have to add a few syntactic constraints.

Automatically Extendable Global Environment

If a toplevel loop is present, then we need to be able to augment the global environment dynamically. We just saw that the form `define` could help augment the set of global variables. You might also think that assignment would suffice for the same task; that is, that assigning a free variable would be equivalent to defining that variable in the global environment if it had not appeared there before. Thus we could directly write this:

```
(let ((name "Nemo"))
  (set! winner (lambda () name))
  (set! set-winner! (lambda (new-name) (set! name new-name)
                                       name )) )
```

With the preceding variation, we would have had to prefix this expression by two absurd definitions, like these:

```
(define winner      'without-tail)
(define set-winner! 'nor-head)
```

Those expressions would have created the two variables, initialized them in any old way, then would have modified them immediately so that they would take on their real value. That technique is annoying because it explicitly makes the defined variables mutable since they have been the object of at least one assignment. For that reason, quite a long time ago, in Scheme [SS78b] there was a `static` form enabling us to write something[2] like this:

```
(let ((name "Nemo"))
  (define (static winner) (lambda () name))
  (define (static set-winner!)
     (lambda (new-name) (set! name new-name)
                        name ) ) )
```

That way, the two global variables would have been co-defined together in a local lexical context without the intervention of assignment.

While assignment makes it possible to create global variables, we risk polluting the global environment by doing so. You might also think that it would be more clever to create the variable only locally; perhaps lexically at the level of the last `let` or, as in [Nor72], at the last dynamically enveloping[3] `prog`. Unfortunately, these ideas spoil referential transparency and forbid the preceding co-definitions.

Hyperstatic Global Environment

There is one more kind of global environment: one where several global variables can be associated with a name but where only forms located after a definition can see it. Let's look again at our favorite example:

```
g                        ;error: no variable g
(define (P m) (* m g))   ;error: no variable g
```

2. Nevertheless, we have to change the syntax of internal `define` forms so that they recognize the local presence of global definitions. That is, (`static` *variable*) is the reference to a global variable, rather than a call to the unary `static` function.

3. TEX does this: a definition made by `\def` disappears when we exit from the current group with `\endgroup`.

```
(define g 10)
(define (P m) (* m g))
(P 10)              → 100
(define g 9.81)
(P 10)              → 100   ;P sees the old g
(define (P m) (* m g))
(P 10)              → 98.1
(set! e 2.78)               ;error: no variable e
```

Here, faithful to the spirit that prefers to close functions in the environment where they are created, the first definition of `P` is statically an error since it references `g` which does not exist at that moment. Another solution would be to allow the definition of `P` but to make any call to `P` an error since `g` had no value a the moment when `P` was created. However, this solution should not be adopted since it delays warning the user and the sooner errors are caught, the better.

The second definition of `P` encloses the fact that `g` has the value 10, and that fact lasts as long as the function `P`. If we want a better approximation of `g` for some reason, we must redefine the function `P` to take account of that new value. The global environment here is managed in a completely lexical way; we call that mode *hyperstatic*, and it's the mode that ML chose. We'll summarize its characteristics in our usual chart.

Reference	x but x must exist
Value	x but x must exist
Modification	`(set!` x ...`)` but x must exist
Extension	`define`
Definition	no

However, that mode causes problems for recursive definitions. We have to be able to define both functions that are simply recursive and groups of functions that are mutually recursive. ML has a keyword—**rec**—to indicate that first kind, and another keyword—**and**—to indicate co-definitions. Those keywords correspond to `letrec` and `let` in Scheme, where those forms already authorize multiple definitions. To make that idea visible, consider the preceding example rewritten this time as a series of nested instances of `let` or `letrec`:

```
g                                       ;error: no variable g
(let ((P (lambda (m) (* m g))))         ;error: no variable g
  (let ((g 10))
    (let ((P (lambda (m) (* m g))))
      (P 10)
      (let ((g 9.81))
        ... ) ) ) )
```

And here's another example, this time in ML, of mutually recursive functions:

```
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)
```

That example obviously corresponds to this:

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

Hyperstatic global environments have the obvious advantage of being able to detect undefined variables statically. Furthermore, if bindings are known to be immutable, these environments also compile very efficiently because they may take advantage of the fact that they know the value. When an error occurs, however, they have the disadvantage that they require a redefinition of everything that follows the erroneous definition.

4.1.3 Assignment of a Predefined Variable

Among the free variables appearing in a program, there are the predefined ones like `car` or `read`. Assigning them is thus our inalienable right, but just what does it mean to assign one? The speed of an implementation frequently depends on hypotheses that are rarely made explicit. In Lisp, we often talk about *inline* functions, that is, compiled and integrated functions; accessors such as `car` or `cdr` are usually inline. Redefining any of them thus causes problems because, once an assignment impinging on `car` is perceived, then all the calls to `car` have to use the current value of `car` rather than its primitive value.

In a hyperstatic interaction loop, that's not a real problem because only future expressions will see this modified `car` function. However, if the interaction loop is dynamic, then to be accurate, we have to recompile every inline call to `car`. That would lead us logically to redefining the function `cadr` since it's probably defined as the composition of `car` and `cdr`. This work would never actually get done since it introduces too much disorder and probably does not correspond to the real intentions of the users.

By the way, Scheme forbids the modification of a global binding from changing the values or the behavior of other predefined functions. In that context, modifying `car` would not be allowed, but defining a new `car` variable would be permitted, though it would probably not change the meaning of `cadr`. Since `map` (comparable to `mapcar` in Lisp) appears in the standard definition of Scheme, a modification of `car` would not perturb it, but since `mapc` is not part of the standard definition, modifying `car` would probably disturb it.

Modifying a global variable often looks like a way to trace or analyze calls to the function that is the value of that global variable. It also seems like a means to correct erroneous or aberrant behavior. However, we advise the impetuous and inexperienced to resist such temptations. Our advice is "Don't touch predefined variables!"

In summary, a hyperstatic global environment behaves logically but it is not as practical as a dynamic global environment for debugging.

4.2 Side Effects

So far, we've seen how a computation in Lisp is represented by an ordered triple of expression, environment, and continuation. As attractive as that triple is, it does not express the meaning of assignment nor of physical modification of data in memory. *Side effects* correspond to alterations in the state of the world and in particular to changes in memory.

The most familiar changes come from physically modifying data in memory, as `set-car!` and input/output functions do. Reading or writing in a stream makes a mark move along to show where the most recent read or write occurred. (That's a side effect!) Writing to a screen leaves an irreparable mark on it—a long lasting side effect—and writing to paper is even more indelible—yet another long term side effect. Requiring a user to type on a keyboard is likewise irreversible. In short, side effects are unfortunately omnipresent in computing. They correspond to sets of instructions for computers operating on registers, to file systems saving information (for example, the marvelous texts and programs that we concoct ourselves) between sessions. Of course, it is possible to imagine living in an ideal world with no side effects, but we would suffer from a kind of computer-autism there since we would not be able to communicate the results of computations. This nightmare won't keep us awake, however, since there is no computer in the world that actually works with no side effects.

We've already seen that assignments can be simulated by boxes. Inversely, can we simulate dotted pairs without dotted pairs? The response is yes, but we do so by using assignment! A dotted pair can be simulated by a function responding to the messages `car`, `cdr`, `set-car!`, and `set-cdr!`[4] like this:

```
(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) )
```

Once again, the simulation is not quite perfect, as [Fel90] points out, because we can no longer distinguish the dotted pairs from normal functions. With that programming practice, we can no longer write the predicate `pair?`. Except for this slight reduction in means (in fact, equivalent to being able to write new types of data), we see clearly that side effects and assignment maintain uneasy relations and that one is easily simulated by the other. Consequently, we have to ban both of them or accept them and their more or less uncontrollable fallout. Whether we choose assignment or physical modification then depends on the context and the properties that we are trying to preserve.

4.2.1 Equality

One inconvenience of physical modifiers is that they induce an alteration in equality. When can we say that two objects are equal? In the sense that Leibnitz used it, two objects are equal if we can substitute one for the other without damage. In programming terms, we say that two objects are equal if we have no means of distinguishing one from the other. The equality tautology says that an object is

4. To avoid confusion with the usual primitives, we'll spell their names with k.

equal to itself since we can always replace it by itself without changing anything. This is a weak notion of equality if we take it literally, but it's the same idea that we apply to integers. Things get more complicated when we consider two objects between which we don't know any relation (for example, when they derive from two different computations, like `(* 2 2)` and `(+ 2 2)`) and when we ask whether they can substitute for each other or whether they resemble each other.

In the presence of physical modifiers, two objects are different if an alteration of one does not provoke any change in the other. Admittedly, we're talking about *difference* here, not equality, but in this context, we'll classify objects in two groups: those that can change and those that are unchanging.

Kronecker said that the integers were a gift from God, so on that logical basis, we'll consider them unchanging: 3 will be 3 in all contexts. Also it's simple for us to decide that a non-composite object (that is, one without constituent parts) will be unchanging, so in addition to integers, we also have characters, Booleans, and the empty list. There aren't any other unchanging, non-composite objects.

For composite objects, such as lists, vectors, and character strings, it seems logical to consider two of them equal if they have the same constituents. That's the idea we generally find underlying the name `equal?` with different variations[5] whether recursive or not. Unfortunately, in the presence of physical modifiers, equality of components at a given moment hardly insures the persistence of equality in the future. For that reason, we'll introduce a new predicate for physical equality, `eq?`, and we'll say that two objects are `eq?` when it is, in fact, the same.[6] This predicate is often implemented in an extremely efficient way by comparing pointers; in that way, it tests whether two pointers designate the same location in memory.

In fact, we have two predicates available: `eq?`, testing the physical identity, and `equal?`, testing the structural equivalence. However, these two extreme predicates overlook the changeability of objects. Two immutable objects are equal if their components are equal. To be sure that two immutable objects are equal at some moment in time insures that they will always be so because they don't vary. Inversely, two mutable objects are eternally equal only if they are one and the same object. In practice, a single idea is emerging here as the definition of equality: whether one object can be substituted for the other. Two objects are equal if no program can distinguish between them.

The predicate `eq?` compares only entities comparable with respect to the implementation: addresses in memory[7] or immediate constants. The conventional techniques of representing objects in the dialects that we're considering do not insure[8] that two equal objects have comparable representations for `eq?`. It follows that `eq?` does not even implement tautology.

A new predicate, known as `eqv?` in Scheme and as `eql` in COMMON LISP, improves `eq?` in this respect. In general terms, it behaves like `eq?` but a little more slowly to insure that two equal immutable objects are recognized as such. The predicate `eqv?` insures that two equal numbers, even gigantic ones represented by

5. See, for example, the functions `equalp` or `tree-equal` in COMMON LISP.
6. This grammatical weirdness expresses the idea that there are not two objects, but only one.
7. For a distributed Scheme on a network of computers, `eq?` has to be able to compare objects located at different sites. In such a case, `eq?` is not necessarily immediate.
8. For example, `(eq? 33 33)` is not guaranteed to return `#t` in Scheme, hence the need for `eqv?`.

bignums, will be compared successfully regardless of their representation in memory.

If we look back at the interpreters that we've given so far, we notice how little mutable data are found in them. With the exception of bindings, everything there was unchanging. In fact, mutable structures are few, and the majority of allocated dotted pairs never submit to physical modifiers such as `set-car!` and `set-cdr!`. In some implementations of ML, but also in COMMON LISP, it's possible to indicate the mutability of fields in an object. We can thus create a constant dotted pair, like this:

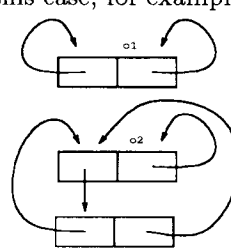
COMMON LISP
```
(defstruct immutable-pair
   (car '() :read-only t)
   (cdr '() :read-only t) )
```

Physically, constant dotted pairs could be allocated in a zone apart, and constants cited in programs could use this type of dotted pair.

In [Bak93], Henry Baker suggested unifying all the equality predicates in a sole `egal` defined like this: if the objects to compare are mutable, then `egal` behaves like `eq?`; otherwise, it behaves like `equal?`. This new predicate insures that two objects can be substituted for each other when they are recognized as equal by `egal`. It's easy to see the utility of this predicate in a parallel world involving the migration of data, as in [QD93, Que94].

Cyclic data present another problem. Comparing them is possible but expensive. If we don't know that the data being compared may be cyclic, then we may fall into a looping `equal?`. And if that problem were not bad enough, how to compare these structures is not obvious either. Consider this case, for example:

```
(define o1 (let ((pair (cons 1 2)))
             (set-car! pair pair)
             (set-cdr! pair pair)
             pair ))
(define o2 (let ((pair (cons (cons 1 2) 3)))
             (set-car! (car pair) pair)
             (set-cdr! (car pair) pair)
             (set-cdr! pair pair)
             pair ))
```



Let's suppose first of all that we have to evaluate `(equal? o1 o1)`. If the implementation of `equal?` begins by testing whether the objects are `eq?` before it begins the structural comparison of their respective fields, then the answer is immediate and positive. In the opposite case, `equal?` will loop forever.[9]

Granted that `(equal? o1 o1)` is a suspicious case, what do you think of `(equal? o1 o2)`? The reply depends once again on knowing whether the dotted pairs composing `o1` and `o2` are mutable or not. If no one can modify them, then in the absence of `eq?`, no one can write a program that distinguishes them. This observation brings us back to the idea that it's better not to compare cyclic structures with `equal?`.

The preceding predicates cover the various kinds of data usually present, with the exception of symbols and functions. Symbols are complex data structures since they are often used by implementations to store information about global variables

9. As I wrote that, I checked four different implementations of Scheme, and two looped. I won't reveal their names since they are fully justified in doing so. [CR91b]

of the same name. Basically, a symbol is a data structure that we can retrieve by name and that is guaranteed unique for a given name.

Symbols often contain a property list—a dangerous addition because a property list is managed by side effects that are perceived globally. Moreover, a property list is usually burdensome, both in storage (since it takes two dotted pairs per property) and in use (since it requires a linear search). For those reasons, *hash tables*[10] are preferable.

4.2.2 Equality between Functions

For functions, our situation may seem desperate. We can say that two functions are equal if their results are equal whenever their arguments are equal, that is, they are undistinguishable. More formally, we can put it this way:

$$f = g \Leftrightarrow \forall x, f(x) = g(x)$$

Unfortunately, comparing two functions that way is an undecidable problem. For that reason, we could refuse to compare functions at all, or we could adopt a more flexible attitude but restrict the problem. Large classes of functions are comparable, and in many cases, we can easily discover that two functions cannot be equal (for example, when they don't even have the same number of arguments).

With those thoughts in mind, we can get an approximate equality predicate for functions by admitting that sometimes its response may be imprecise or even erroneous. Of course, we can rely on such an approximate predicate only if we understand clearly where and how it is imprecise.

Scheme, for example, defines `eqv?` for functions in the following way: if we must compare the functions `f` and `g`, and if there exists an application of `f` and `g` to the same arguments (where "the same arguments" is determined by `eqv?`) such that the results are different, then the functions `f` and `g` are not `eqv?`. Once again, we find ourselves considering difference rather than defining equality.

Let's look at a few examples now. The comparison `(eqv? car cdr)` should return false since their results are manifestly different, especially on an example like `(a . b)`.

It should also be obvious that `(eqv? car car)` returns true since equality must surely handle tautology correctly. However, it's false in certain dialects where that form is equivalent to `(eqv? (lambda (x) (car x)) (lambda (x) (car x)))` because the function `car` can be inline. In fact, R^{4}RS does not specify what `eqv?` should return when it is used to compare predefined functions like `car`.

How can we compare `cons` and `cons` without invoking tautology? The function `cons` is an allocator of mutable data, and thus, even if the arguments seem comparable, the results are not necessarily equal since they are allocated in different places. Thus we're right to demand that `cons` should not be equal to `cons` and thus `(eqv? cons cons)` should return false since it is false that `(eqv? (cons 1 2) (cons 1 2))`!

Now is it really true that `car` should be equal to `car` as we argued in the previous paragraphs? In a language without types, `(car 'foo)` raises an error and signals an exception, but then it's left up to the care of the local error handler, a

10. In my humble opinion, their invention was one of the greatest discoveries of computer science.

mechanism with unpredictable behavior that may vary wildly. For that reason, we cannot be sure that that call always returns favorably comparable results. The same observations hold for any partial function used outside its domain of definition.

How do we get out of this situation? Different languages try different means. Because of their typing which lets them detect all the places where comparisons between functions might be made, some dialects of ML forbid such situations purely and simply since we cannot and should not compare functions. In its semantics, Scheme makes **lambda** a closure constructor. Thus each **lambda** form allocates a new closure somewhere in memory. That closure has a certain address, so functions can be compared by a comparison of addresses, and the only case where two functions are the same (in the sense of **eqv?**) is when they are one and the same closure.

This behavior prevents certain improvements known in λ-calculus. Let's consider the function cleverly named **make-named-box**. It takes only one argument (a message), analyzes it, and responds in an appropriate way. This is one possible way of producing objects, and it's the one that we adopt for coding the interpreter in this chapter.

```
(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))) ) ) )
```

That function creates an anonymous box and then gives it a name. The closure gotten by means of the message **type** does not depend on any of the local variables. Consequently, we could rewrite the entire function like this:

```
(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) ) ) ) ) ) )
```

This one differs from the preceding version because the closure **(lambda () 'named-box)**, for example, is allocated only once whereas earlier, it was allocated every time it was needed.

But then what's the value of the following comparison?

```
(let ((nb (make-named-box 'foo 33)))
  (compare? (nb 'type) (nb 'type)) )
```

Actually the question is badly phrased because it has meaning only if we specify the predicate for comparison. If we use `eq?`, then the exact number of allocations of `(lambda () 'named-box)` will play a role, whereas if we use `egal`, then the question is meaningless, and the final value is always true. From this, you can see that if a language includes a physical comparer, that comparer will make it possible to discern whether objects might possibly be equal.

When `lambda` is an allocator, we associate an address in memory with all the closures that it creates. Two closures having the same address are merely one and the same object and thus equal. Accordingly, in Scheme, we have this:

```
(let ((f (lambda (x y) (cons x y))))
  (eqv? f f) )     → #t
```

Since it compares the same object in memory, the predicate `eqv?` will return true even if it is false that `(eqv? (f 1 2) (f 1 2))`.

As you see, comparing functions is an activity beset by obstacles for which multiple points of view might be adopted, depending on the properties that we want to preserve. Keeping the mathematical aspect of equality of functions is practically out of the question. Depending on the implementation, many ways of using comparisons may be proscribed. Even though this very book contains such a comparison in the implementation of MEROONET, [see p. 447] we still offer the advice that, as far as possible, it is better to avoid comparing functions at all.

4.3 Implementation

For once, the interpreter that we'll explain in this chapter uses closures only for its data structures. Everything will be coded by `lambda` forms and by sending messages. All the objects are thus closures based on code functions similar to `(lambda (msg) (case msg ... ))` like we saw in a few of the preceding examples. Some messages will be standard, like `boolify` or `type`. `boolify` associates a value with its equivalent truth value (that is, `#t` or `#f` for the purposes of conditionals). `type`, of course, returns its type.

Our chief problem is to find a method to define side effects. Formally, we say that a variable references a binding which is associated with a value. More prosaically, we say that a variable points to a box (an address) that contains a value. Memory is merely a function associating addresses with the values contained in those addresses. The environment is another function associating these addresses with variables. Those conventions all seem natural until you try to simulate their implementation.

Memory must be visible everywhere. We could make it a global variable, but that is not a very elegant solution, and we have already seen all the problems that a global environment entails. Another solution is to make all the functions see the memory. It would then suffice for all the functions to receive the memory as an argument and pass it along to others, possibly after modifying it. We'll actually adopt this way of doing it, and that choice forces us to consider a computation now as a quadruple: *expression*, *environment*, *continuation*, and *memory*. Those four components are named `e`, `r`, `k`, and `s`. To make sure that memory circulates freely,

continuations will receive not only the value to transmit but also the resulting memory state.

As usual, the function `evaluate` will syntactically analyze its argument and call the appropriate function. We'll keep the conventions for naming variables from the previous chapters, and we'll add the following:

`e`, `et`, `ec`, `ef` ...	expression, form
`r` ...	environment
`k`, `kk` ...	continuation
`v`, `void` ...	value (integer, pair, closure etc.)
`f` ...	function
`n` ...	identifier
`s`, `ss`, `sss` ...	memory
`a`, `aa`	address (box)

Since the number of variables has increased, we'll adopt the discipline of always mentioning them in the same order: `e`, `r`, `s`, and then `k`.

Here's the interpreter:

```
(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 Conditional

A conditional introduces an auxiliary continuation that waits for the value of the condition.

```
(define (evaluate-if ec et ef r s k)
  (evaluate ec r s
    (lambda (v ss)
      (evaluate ((v 'boolify) et ef) r ss k) ) ) )
```

The auxiliary continuation takes into consideration not only the Boolean value `v` but also the memory state `ss` resulting from the evaluation of the condition. In effect, it is legal to have side effects in the condition, as the following expression shows:

```
(if (begin (set-car! pair 'foo)
           (cdr pair) )
    (car pair) 2 )
```

Any modification that occurs in the condition has to be visible in the two branches of the conditional. The situation would be quite different if we defined the conditional like this:

```
(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 ) ) ) )
```

In that latter case, the memory state present at the beginning of the evaluation of the condition would be restored. That would be characteristic of a language that supports backtracking in the style of Prolog. [**see Ex. 4.4**]

To take into account the fact that every object is coded by a function, True and False must not be the `#t` and `#f` of the implementation language. Since every object can also be considered as a truth value, we will assume that every object can receive the message `boolify` and will then return one of the λ-calculus style combinators, `(lambda (x y) x)` or `(lambda (x y) y)`.

4.3.2 Sequence

Though it's not essential to us, the definition of a sequence shows clearly just how sequencing is treated in the language. It also highlights an intermediate continuation. For simplicity, we'll assume that sequences include at least one term.

```
(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) ) )
```

In such a bare form, you can see how little regard there is for the value of `void` and the final call to `evaluate` when there is only one form in a sequence.

4.3.3 Environment

The environment has to be coded as a function and must transform variables into addresses. Similarly, the memory is represented as a function, but one that transforms addresses into values. Initially, the environment contains nothing at all:

```
(define (r.init id)
  (wrong "No binding for" id) )
```

Now how do we modify an environment or even memory without a physical modifier and without assignment? To modify a memory state is to change the value associated with an address. We could thus define the function `update` so that it modifies memory when we offer it an address and a value, like this:

```
(define (update s a v)
  (lambda (aa)
    (if (eqv? a aa) v (s aa)) ) )
```

The meaning is apparent if we remember that a memory state is represented, as we said, by a function that transforms addresses into values. The form `(update`

`s a v)` returns a new memory state that faithfully resembles the `s` except that at the address `a` the associated value is now `v`; for everything else, we see `s`. We generalize **update** to take multiple addresses and multiple values, like this:

```
(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 ) )
```

The function **update** can also be applied to extending environments. In ML, that would be a polymorphic function. Keeping in mind the kind of comparison that **update** carries out, we'll code addresses as objects that are eminently comparable: integers. To compare variables, we'll compare their names. In Scheme, the function **eqv?** will thus be convenient in both cases.

4.3.4 Reference to a Variable

Consequently, the value of a variable is expressed simply, like this:

```
(define (evaluate-variable n r s k)
  (k (s (r n)) s) )
```

The form `(r n)` returns the address where the value of the variable is located (that is, the value of `n`). The contents of this address is searched for in memory and given to the continuation. Since searching for the value of a variable is a non-destructive process, memory will not be modified that way and will be transmitted as such to the continuation.

4.3.5 Assignment

Assignment requires an intermediate continuation.

```
(define (evaluate-set! n e r s k)
  (evaluate e r s
    (lambda (v ss)
      (k v (update ss (r n) v)) ) ) )
```

After the evaluation of the second term, its value and the resulting memory are provided to the continuation to update the contents of the address associated with the variable. That is, a new memory state is constructed, and that new state will be provided to the original continuation of the assignment. By the way, this assignment returns the assigned value.

Here you can see why making memory a function was a good idea. Doing so lets us represent modifications without any side effects. In more Lispian terms, this way of representing memory turns it into the list of all modifications that it has undergone. Compared to real memory, this representation is manifestly superfluous, but it enables us to handle various instances of memory simultaneously. **[see Ex. 4.5]**

4.3.6 Functional Application

A functional application consists of evaluating all terms. Here we'll choose left to right order.

```
(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" (car v*)) ) ) ) ) ) )
```

Here, the function usually named **evlis** becomes local, known under the name **evaluate-arguments**. It evaluates its arguments in order and organizes their values into a list. The function (the value of the first term of the functional application) is then applied to its arguments accompanied by the memory and the continuation of the call. There again, you can see how concise the program is.

Since the function is represented by a closure that responds at least to the message **boolify**, a new message—**behavior**—will extract its functional behavior, that is, the way it calculates.

4.3.7 Abstraction

To simplify things, let's assume for the moment that the special form **lambda** creates only functions with fixed arity. Two effects come together here—an allocation in memory and the construction of a first class value—as a closure. Creating a function involves constructing an object that, as you might expect, has to be allocated somewhere in memory. In other words, it is necessary for memory to be modified when a function is created. To that end, we furnish two things to the utility **create-function**: an address and the behavior of the function to create. The behavior we furnish is the same that the message **behavior** extracted before, during the functional application.

```
(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 ) ) ) )
```

When a function constructed by the special form `lambda` is called on one of its arguments, its behavior stipulates that, in current memory at the moment of the call, it must allocate as many new addresses as it has variables to bind; then it must initialize these addresses with the associated values and finally pursue the rest of its computations.

```
(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 ) ) ) )
```

If the addresses were allocated at another time, for example, at the time the function was created, we would get behavior more like that of Fortran, forbidding recursion. In effect, every recursive call to the function uses those same addresses to store the values of variables, thus making only the last such values accessible. The purpose for this variation is that there is no dynamic creation of functions in Fortran, and consequently these addresses can be allocated during compilation, thus making function calls faster but thereby forbidding recursion, the real strength of functional languages.

4.3.8 Memory

Memory is represented by a function that takes addresses and returns values. It must also be possible to allocate new addresses there, new addresses that are guaranteed free; that's the purpose of the function `allocate`. As arguments, it takes a memory and the number of addresses that it must reserve there; it also takes a third argument: a function (a continuation) to which it will give the list of addresses that it has selected in memory plus the new memory where these addresses have been initialized to the "uninitialized" state. The function `allocate` handles all the details of memory management and notably the recovery of cells that is, garbage collection. Fortunately, it's simple to characterize this function if we assume that memory is infinitely large; that is, if we assume that it's always possible to allocate new objects.

```
(define (allocate n s q)
  (if (> n 0)
```

```
(let ((a (new-location s)))
  (allocate (- n 1)
            (expand-store a s)
            (lambda (a* ss)
              (q (cons a a*) ss) ) ) )
(q '() s) ) )
```

The form **new-location** searches for a free address in memory. This is a real function in the sense that the form **(eqv? (new-location s) (new-location s))** always returns True. We associate the highest address used so far with each memory; that address will be used when we search for a free address by means of **new-location**. To mark that an address has been reserved, we extend memory by **expand-store**.

```
(define (expand-store high-location s)
  (update s 0 high-location) )
(define (new-location s)
  (+ 1 (s 0)) )
```

Initial memory doesn't contain anything, but it defines the first free address. Memory is thus a closure responding to a certain message when it involves determining a free address or responding to messages coded as integers (that is, addresses) when we want to know the contents of memory. We'll unify these two kinds of messages by assuming that the address 0 contains the address most recently used.

```
(define s.init
  (expand-store 0 (lambda (a) (wrong "No such address" a))) )
```

4.3.9 Representing Values

We've decided to represent all values that the interpreter handles by functions that send messages. These values are the empty list, Booleans, symbols, numbers, dotted pairs, and functions. We'll look at each of these kinds of data in turn.

All values will thus be created on a skeleton function that responds to at least two messages: *(i)* **type** for requesting its type; *(ii)* **boolify** for converting it to a truth value. Other messages exist for specific types of data. The skeleton that serves as the backbone of all these values will thus be this:

```
(lambda (msg)
  (case msg
    ((type)    ...)
    ((boolify) ...)
    ... ) )
```

There is only one unique empty list, and according to R^{4}RS, it is a legal representation of True, so we use this:

```
(define the-empty-list
  (lambda (msg)
    (case msg
      ((type)    'null)
      ((boolify) (lambda (x y) x)) ) ) )
```

The two Boolean values are created by **create-boolean**, like this:

```
(define (create-boolean value)
  (let ((combinator (if value (lambda (x y) x) (lambda (x y) y))))
    (lambda (msg)
      (case msg
        ((type)    'boolean)
        ((boolify) combinator) ) ) ) )
```

Symbols should respond to the specific message **name** that extracts their name represented as a symbol in the defining Scheme.

```
(define (create-symbol v)
  (lambda (msg)
    (case msg
      ((type)    'symbol)
      ((name)    v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Numbers will have **value** as their specific message, like this:

```
(define (create-number v)
  (lambda (msg)
    (case msg
      ((type)    'number)
      ((value)   v)
      ((boolify) (lambda (x y) x)) ) ) )
```

Functions will respond to the messages **behavior** and **tag**. Of course, **behavior** extracts their behavior; **tag** indicates the address to which they have been allocated.

```
(define (create-function tag behavior)
  (lambda (msg)
    (case msg
      ((type)     'function)
      ((boolify)  (lambda (x y) x))
      ((tag)      tag)
      ((behavior) behavior) ) ) )
```

The last case we have to cover is that of dotted pairs. A dotted pair indicates two values that are both susceptible to modification by physical modifiers. For that reason, we're going to represent dotted pairs by a pair of addresses: one will contain the **car** of the dotted pair while the other will contain its **cdr**. This choice of representation may seem strange since a dotted pair is conventionally represented by a box of two contiguous values.[11] Thus only one address is needed to indicate the pair. This way of coding does not do justice to that implementation technique consisting of storing the **car** and **cdr** in two different arrays but at the same index in each array; a dotted pair is thus represented by such an index. According to that way of coding, two different addresses are thus associated with each dotted pair.

11. Very serious studies, such as [Cla79, CG77], have shown that it's better for cdr (rather than car) to be directly accessible without a supplementary displacement. Another reason to place cdr first is that pairs implement lists and thus inherit from the class of linked objects which define only one field: cdr.

To simplify the allocation of lists, we'll use the following two functions that take a list and memory, and then allocate the list in that memory. Since that allocation modifies the memory, a third argument (a continuation) will be called finally with the resulting memory and the allocated list.

```
(define (allocate-list v* s q)
  (define (consify v* q)
    (if (pair? v*)
        (consify (cdr v*) (lambda (v ss)
                            (allocate-pair (car v*) v ss q) ))
        (q the-empty-list s) ) )
  (consify v* q) )
(define (allocate-pair a d s q)
  (allocate 2 s
   (lambda (a* ss)
     (q (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 A Comparison to Object Programming

The skeleton closure that we chose to represent values for this interpreter enables those values to respond to multiple messages, and in that sense, those values resemble the objects we used in the preceding chapter. Nevertheless there are several important differences between these objects coded by closures and those of MEROONET. The objects coded by closures contain their methods inside themselves; it is not possible to add new methods to them. The ideas of class and subclass remain virtual concepts since they are not implemented by these values. However, generic functions make it possible to add behavior to objects from the exterior without even requiring their cooperation; generic functions support not only methods, but also multimethods. Finally, the idea of a subclass makes it possible to share the structure of objects without useless repetition of their common characteristics. These various qualities shouldn't make you think of objects as *poor man's closures*, like some Scheme users suggest. In fact, the objects of the previous chapter have many more qualities than those of this chapter, but you can see an interesting defense of these latter in [AR88].

4.3.11 Initial Environment

As usual, we'll introduce two different kinds of syntax to put predefined variables into the initial environment (here, into the initial memory). With every global

variable, we'll associate an address containing its value. The syntax of **definitial** will thus allocate a new address and fill it with the appropriate value.

```
(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)) ) ) ) ) )
```

The syntax of **defprimitive** will be built on top of **definitial**; it will define a function of which the arity will be checked.

```
(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) ) ) ) ) ) ) ) ) )
```

As we've become accustomed to doing, we will enrich the initial environment with the Boolean variables **t** and **f** and with the empty list **nil**, like this:

```
(definitial t (create-boolean #t))
(definitial f (create-boolean #f))
(definitial nil the-empty-list)
```

We'll give two examples of predefined functions, one a predicate and the other arithmetic. Their arguments are extracted from their representation, and then the final result is repackaged as it should be, like this:

```
(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 "<= require numbers") ) )
  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 "* require numbers") ) )
  2 )
```

4.3.12 Dotted Pairs

For the first time among all the interpreters that we've shown so far, the dotted pairs of the Scheme we are defining will not be represented by the dotted pairs of the definition Scheme.

Because of the function `allocate-pair`, constructing a dotted pair is simple enough.

```
(defprimitive cons
  (lambda (v* s k)
    (allocate-pair (car v*) (cadr v*) s k) )
  2 )
```

Reading or modifying fields is also simple, like this:

```
(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 )
```

All values that can be manipulated are coded so that their type can be inspected. Since all objects know how to answer the message `type`, it is simple to write the predicate `pair?`. We'll use `create-boolean` to convert a Boolean from the definition Scheme into a Boolean of the Scheme being defined, like this:

```
(defprimitive pair?
  (lambda (v* s k)
    (k (create-boolean (eq? ((car v*) 'type) 'pair)) s) )
  1 )
```

4.3.13 Comparisons

One of the goals of this interpreter is to specify the predicate for physical comparisons: `eqv?`. That predicate physically compares two objects as well as numbers. That predicate first compares the types of the two objects, then, if they agree, it specifically compares the two objects. Symbols must have the same name to pass the comparison. Booleans and numbers must be the same. Dotted pairs or functions must be allocated to the same address(es).

```
(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) ) )
                ((symbol)
                 (eq? ((car v*) 'name) ((cadr v*) 'name)) )
                ((number)
                 (= ((car v*) 'value) ((cadr v*) 'value)) )
                ((pair)
                 (and (= ((car v*) 'car) ((cadr v*) 'car))
                      (= ((car v*) 'cdr) ((cadr v*) 'cdr)) ) )
                ((function)
                 (= ((car v*) 'tag) ((cadr v*) 'tag)) )
                (else #f) )
               #f ) )
          s ) )
    2 )
```

In fact, the only reason for giving a function an address (other than the one we just mentioned that functions—or more precisely, their closures—are data structures allocated in memory) is that we can then compare functions by means of **eqv?**. Functions are projected onto their addresses, which are entities that are easy to compare since they are merely integers. Notice that comparing dotted pairs does not necessitate any inspection of their contents but only the addresses associated with them. In consequence, comparison is a function with constant cost compared to **equal?**.

The definition of **eqv?** might seem generic since we have defined it as a set of methods for disparate classes. In fact, a clever representation of data let's us cut through type-testing and go directly to comparing the implementation address of objects; we can often reduce that activity to a simple instruction, except possibly for *bignums*.

4.3.14 Starting the Interpreter

Now we'll get right to the interpreter. The toplevel loop is re-invoked in the continuation that it gives to **evaluate**, still insuring the diffusion of the memory state acquired during the most recent interaction.

```
(define (chapter4-interpreter)
  (define (toplevel s)
    (evaluate (read)
              r.global
              s
              (lambda (v ss)
                (display (transcode-back v ss))
                (toplevel ss) ) ) )
  (toplevel s.global) )
```

The basic problem with this interpreter is that for the first time in this book, the data of the Scheme being interpreted are quite different from the underlying Scheme. This difference is particularly noticeable for dotted pairs, that is, for lists

that serve internally, for example, to organize arguments submitted to a function. Since the dotted pairs of the two levels (defining and defined languages) are no longer equivalent, we resort to **transcode-back** for decoding the final value that we get. That function runs the value through certain memory state and transforms it into a value of the definition Scheme so that we can then print it. We could equally well have chosen to print it directly without passing by the **display** function of the underlying Scheme.

```
(define (transcode-back v s)
  (case (v 'type)
    ((null)     '())
    ((boolean)  ((v 'boolify) #t #f))
    ((symbol)   (v 'name))
    ((string)   (v 'chars))
    ((number)   (v 'value))
    ((pair)     (cons (transcode-back (s (v 'car)) s)
                      (transcode-back (s (v 'cdr)) s) ) )
    ((function) v) ;why not ?
    (else       (wrong "Unknown type" (v 'type))) ) )
```

4.4 Input/Output and Memory

We haven't yet talked about input/output functions. If we limit ourselves to a single input stream and a single output stream (that is, to the two functions **read** and **display**), then the problem could be treated precisely by adding two supplementary arguments to the interpreter to represent these two streams. The input stream would contain all that would be read, while the output stream would be made up of all that would be written there. We could even assume that the output stream would be the only observable response from the interpreter.

The output stream could be represented by a list of pairs (memory, value)—the ones that we provided one-by-one to the function **transcode-back**. But then what would the input stream be? The question is a subtle one because the values that will be read might contain dotted pairs that could be modified physically themselves. We could, indeed, write **(set-car! (read) 'foo)** and then read **(bar hux)**. That observation indicates that the expression read must be installed in the memory current at the moment that **read** is called. The function **transcode** will thus take a value of the underlying Scheme, a memory, and a continuation, and then will call this continuation on the transcoded value and the new memory state in which it has been installed.

```
(define (transcode v s q)
  (cond
   ((null? c)    (q the-empty-list s))
   ((boolean? c) (q (create-boolean c) s))
   ((symbol? c)  (q (create-symbol c) s))
   ((string? c)  (q (create-string c) s))
   ((number? c)  (q (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 q) ) ) ) ) ) ) )
```

We won't go any further with this variation because it necessitates two more arguments (to indicate the input and output streams) in all the functions we've presented in this chapter.

4.5 Semantics of Quotations

Perhaps you've noticed the omission of quotations from the current interpreter. The form **quote** was always "biblically" simple in the preceding chapters because the values of the definition Scheme and the Scheme being defined were identical. Since that identity no longer holds, we can no longer write things like this:

```
(define (evaluate-quote v r s k)                    WRONG
  (k v s) )
```

We would commit an error there by confusing **v** as an element of the program (the element which *defines* the value to quote) with the value of the form. In other words, **v** is not the value to return but rather the definition of it. That distinction was not apparent in the previous interpreters because of their transparent coding. We must thus make use of **transcode** to transform **v** into a value in the Scheme being defined, like this:

```
(define (evaluate-quote c r s k)
  (transcode c s k) )
```

This definition is *compositional* because computing the value returned by the quotation depends only on the arguments provided to **evaluate-quote**. At the same time, it introduces a breach between the current program in Lisp or Scheme. Consider only the expression `(quote (a . b))`. By definition, as we have given it, it is *exactly* equivalent to `(cons 'a 'b)`.[12] When we quote a composite object, like the dotted pair here `(a . b)`, we induce the construction of such an object in current memory. Thus quotation is a shortcut conveniently expressing the idea that a value is being constructed. That fact excludes the possibility of the following expression returning True:

```
(let ((foo (lambda () '(f o o))))
  (eq? (foo) (foo)) )
```

In that example, the local function **foo** synthesizes new values at each call; the futures of those values are independent of one another; those values are different in the sense of **eq?**. If we always want True from that computation, then we need to hack **evaluate-quote** so that it always returns the same value once such a value has been chosen; that is, we need to make a memo-function. One reason for wanting such a device is that, in the absence of side effects, (that is, in absence of **eq?**), it is

12. In more concise terms, `'(a . b)≡`‘`(,'a . ,'b)`.

not necessary to construct the same value over and over again, and besides, doing so would be quite costly. Indeed, we can do without that completely.

Of course, in the presence of side effects, the situation is completely different, as you can see from the following:

```
(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) )) ) ) ) )
```

However, the preceding memo-function has re-introduced an assignment, starting from a side effect, something for which we would like to reduce the need. Another solution is to transform the initial program in such a way to regroup all the quotations into one place where they will be evaluated only once: in the header of the program. Then every time that a quotation appears, it will be replaced by a reference to a variable which has already been correctly initialized. Accordingly, we'll transform the program like this:

```
(let ((foo (lambda () '(f o o))))          (define quote35 '(f o o))
  (eq? (foo) (foo)) )              ⇝       (let ((foo (lambda () quote35)))
                                             (eq? (foo) (foo)) )
```

The program transformed that way will regain the usual semantics of Lisp without our necessarily touching the definition of **evaluate-quote**.

If we want to get rid of quoting composite objects totally, we could follow up the transformation we explained earlier and make explicit all the successive allocations for constructing quoted values. In that way, we get this:

```
(define quote36 (cons 'o '()))
(define quote37 (cons 'o quote36))
(define quote38 (cons 'f quote37))
(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

However, that's not the end of this issue since we must also insure that quoted symbols of the same name still correspond to the same value. For that reason, we'll continue the transformation like this:

```
(define symbol40 (string->symbol "f"))
(define symbol39 (string->symbol "o"))
(define quote36 (cons symbol39 '()))
(define quote37 (cons symbol39 quote36))
(define quote38 (cons symbol40 quote37))
(let ((foo (lambda () quote38)))
  (eq? (foo) (foo)) )
```

We'll stop there since we can consider strings as primitive objects. We can translate them as such in assembly language or in C, and it would be too costly to

translate them into their ultimate components: characters!

In the end, we can accept a compositional definition of quoting since, by transforming the program, we can revert to our usual habits. However, we might scrutinize these bad habits that depend on the fact that programs are often read with the function **read** and that thus the expression that appears in a **quote** and that specifies the immediate data to return is coded with the same conventions: same dotted pairs, same symbols, etc. Natural laziness thus impinges on the interpreter to use this same value and thus to return it every time it's needed. In doing so, we share it with all the receivers of the quotation which leads to the misunderstandings that have long been the delight of Lispers of the old school. Consider, for example, the following expressions:

```
(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)   → ?
```

When those expressions are interpreted, there's a strong possibility that the return value will not be **#t**, and they will provoke an error. Besides, if we print the definition of the function (or if we had made the closed variable **vowels** a global variable instead) then we would see that it changes into this:

```
(define vowel1<=
  (let ((vowels '(#\a #\e)))
    (lambda (c1 c2)
      (memq c2 (memq c1 vowels)) ) ) )
```

By doing so, we make part of the program disappear! The same technique could inversely make the value of the variable **vowels** grow instead of diminishing it. This "memo-visceral" effect happens only with the interpreter; it has no meaning if we compile the preceding program. In effect, if the global variable **vowel<=** is immutable, a call to **(vowel<= #\o #\u)** where the function and all its arguments are known, can be replaced by **#t**. That phenomenon is known as *constant folding*, a technique generalized by partial evaluation, as in [JGS93].

The compiler might also decide to transform the preceding definition after analyzing the possible cases for **c1**, like this:

```
(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) ) ) )
```

This new equivalent form does not guarantee that **(eq? (cdr (vowel<= #\a #\e)) (vowel<= #\e #\i))** since the results do not come from the same quotation. In Lisp, it is not even guaranteed that the value of **(eq? (cdr (vowel<=**

`#\a #\e)) (vowel<= #\e #\i))` will always be False because compilers usually retain the right to coalesce constants (that is, to fuse quotations) in order to make them take less space. Compilers might thus transform the preceding definition into this:

```
(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) ) ) )
```

This kind of transformation has an impact on shared quotations, and you can see the impact when you use `eq?`. One simple solution to all these problems is to forbid the modification of the values of quotations. Scheme and COMMON LISP take that approach. To modify the value of a quotation has unknown consequences there.

We could also insist that the value of a quotation must really be immutable, and to do so, we would define quotation like this:

```
(define (evaluate-immutable-quote c r s k)
  (immutable-transcode c s k) )
(define (immutable-transcode c s q)
  (cond
   ((null? c)    (q 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 q ) ) ) ) ) )
   ((boolean? c) (q (create-boolean c) s))
   ((symbol? c)  (q (create-symbol c) s))
   ((string? c)  (q (create-string c) s))
   ((number? c)  (q (create-number c) s)) ) )
(define (allocate-immutable-pair a d s q)
  (allocate 2 s
   (lambda (a* ss)
     (q (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) ) ) )
```

With that definition, any attempt to corrupt a quotation will be detected. This way of distinguishing mutable from immutable dotted pairs can be extended to character strings, and (as in Mesa) we could distinguish modifiable *strings* from unmodifiable *ropes*.

In partial conclusion, it would be better not to attempt to modify values returned by quotations. Both Scheme and COMMON LISP make that recommendation. Even so, we're not yet at the end of the problems posed by quoting. We've already mentioned the technique of coalescing quotations and indicated how that technique can insure equal quotations constructing values that are physically equal according to **eq?**. Conversely, it could happen that we have physical equalities in quotations that we would like to sustain in their values. There are at least two ways to specify such physical equalities: macros and special handling in the reader.

In COMMON LISP, macro-characters carry out special programmable treatments when certain characters are read. For example, **#.***expression* "reads" the value of *expression*. In other words, *expression* is read and evaluated[13] on the fly, and its value is reputed to be whatever was read. Accordingly, we can write this:

COMMON LISP

```
(define bar (quote #.(let ((list '(0 1)))
                       (set-cdr! (cdr list) list)
                       list )))
```

That expression initializes the variable **bar** with a circular list of 0 and 1, alternately. The quotation created that way defines a value that is physically equal to its **cddr**. CLtL2 [Ste90] insures that it will be equal to its **cddr** at execution. You can imagine that this practice makes the transformation of quotations we explained earlier somewhat more complicated because we now have to take into account the cycles for reconstructing them.

The preceding cycle cannot be constructed in Scheme because there we do not have character-macros. Nevertheless, we have macros that serve the same purpose. The following macros cannot be written with **define-syntax** so we use **define-abbreviation**, a macro for defining macros that we'll analyze in Chapter 9. **[see p. 311]**

```
(define-abbreviation (cycle n)
  (let ((c (iota 0 n)))
    (set-cdr! (last-pair c) c)
    `(quote ,c) ) )
(define bar (cycle 2))
```

In that way, we construct a circular list by means of a number that is known during macro-expansion and that appears in a quotation. We could not have written that quotation by hand because it is not possible to make Scheme read

13. We can't say how. That is, we don't know in which environment nor with which continuation.

cyclic data. That's possible in COMMON LISP with the aid of the character-macros `#n=` and `#n#`[14]:

```
(define bar #1=(0 1 . #1#))
```

The current semantics of Scheme seem to exclude such cycles because they correspond to definitions of values that we could not have written by hand. The only legal quotations are, in fact, those that we can write.

In Chapter 9, we'll go more deeply into the many problems posed by macros. [see p. 311]

4.6 Conclusions

Following this chapter, we could give a prodigious amount of advice to Schemers and other Lispers. The precise nature of global variables is complicated, even more so if we consider the problems of modules. Various ideas of equality are subtle and far from widely shared. Finally, quotation is not as simple as it first appears; we should abstain from convoluted quotations.

4.7 Exercises

Exercise 4.1 : Give a functional definition (without side effects) of the function `min-max`.

Exercise 4.2 : The dotted pairs that we simulated with closures use symbols as messages. Disallow the operations `set-car!` and `set-cdr!`, and rewrite the simulation by using only λ-forms.

Exercise 4.3 : Write a definition of `eq?` comparing dotted pairs with the help of `set-car!` or `set-cdr!`.

Exercise 4.4 : Define a new special form of syntax (`or` α β) to return the value of α if it's True; otherwise, undo all the side effects of the evaluation of α and return what β returns.

Exercise 4.5 : Assignment as we defined it in this chapter returns the value that has just been assigned. Redefine assignment so that its value is the value of the variable before assignment.

Exercise 4.6 : Define the functions `apply` and `call/cc` for the interpreter in this chapter.

14. In the example, it's necessary to read an object that is part of itself. Consequently, the function `read` must be intelligent enough to allocate a dotted pair and then fill it. You see there the usual paradoxes involving self-referential objects, like `#1=#1#`.

Exercise 4.7 : Modify the interpreter of this chapter to introduce n-ary functions (that is, functions with a dotted variable).

5
Denotational Semantics

AFTER a brief review of λ-calculus, this chapter unveils denotational semantics in much of its glory. It introduces a new definition of Lisp—this time a denotational one—differing little from that of the preceding interpreter but this time associating each program with its meaning in the form of a respectable mathematical object: a term from λ-calculus.

What exactly is a program? A *program* is the description of a computing procedure that aims at a particular result or effect.

We often confuse a program with its executable incarnations on this or that machine; likewise, we sometimes treat the file containing the physical form of a program as its definition, though strictly speaking, we should keep these distinct.

A program is expressed in a *language*; the definition of a language gives a *meaning* to every program that can be expressed by means of that language. The meaning of a program is not merely the value that the program produces during execution since execution may entail reading or interacting with the exterior world in ways that we cannot know in advance. In fact, the meaning of a program is a much more fundamental property, its very essence.

The meaning of a program should be a mathematical object that can be manipulated. We'll judge as sound any transformation of a program, such as, for example, the transformation by boxes that we looked at earlier [see p. 115], if such a transformation is based on a demonstration that it preserves the meaning of every program to which it is applied. The meaning of a program must be a respectable mathematical object in that the meaning must not be ambiguous and it must be susceptible to the tools that have developed over centuries, even millenia, of mathematical practice. To give a meaning to a program is to associate it with an object from another space, a space more obviously mathematical. For example, it would be judicious, even clever, for the the program defining `factorial` [see p. 27] to be exactly the usual mathematical factorial because then we would know that this definition does, indeed, compute the factorial, and we would also know whether it variants like `meta-fact` [see p. 63] were really equivalent.

If we want to associate meaning with a program, then we have to search for a mathematical equivalent, but this search forces us to know the properties of the programming language that we're using. The problem is thus to associate a programming language with a method that gives meaning to every program written

in that language. In that sense, we speak of the *semantics* of a programming language. The semantics of a programming language has more than one use: the semantics enables us to understand a language so that we can implement it, so that we can prove the validity of transformations in it, so that we can compare its characteristics with other languages, and so forth. Indeed, the uses of semantics are numerous, but semantics are not unique, and diverse methods exist.

The most venerable method of defining a language is surely to choose a *reference implementation.* Then when we have a question about the language, such questions as, for example, the value or the effects of such and such a program, then we submit the issue to the reference implementation, and we accept its answer as if it were an oracle. The difficulty here, though, is that we can hardly build a theory on the reference implementation, certainly not if we have to treat it like a black box. In contrast, if we're really interested in the implementation and consequently try to open the black box, as it were, we'll find ourselves face to face with a program written in a certain language, and our ignorance of that language or any ambiguities in it put us squarely back in the problem of meaning again.

Another approach exploits the idea of a *virtual machine.* With it, we don't escape entirely from the problem we just mentioned, but we divide it into two distinct parts. In one part, the language is defined in terms of a virtual machine having a certain architecture and instruction set. All instructions of the language are defined by a certain group of instructions in the virtual machine. If we know how the virtual machine works, then we understand the language. In the other part, the virtual machine itself is written in an *ad hoc* formalism making it possible to implement the virtual machine on any reasonable computer.

Many languages have been defined in that way, from PL1 (with VDM) to Le-Lisp (and LLM3 [Cha80]), PSL [GBM82] or even Gambit above PVM [FM90].

The main difficulty is how to elaborate the virtual machine since it has to be simultaneously clear in its intention, easy to use, and trivial to implement. To do all that, we have the choice of machines with stack(s), register(s), driven by trees or graphs, etc. (This scenario corresponds to an idealized version of programming in assembly language where the designer would have the freedom to define his or her own machine.) This technique makes it possible to compare programs by looking at their translations into the virtual assembler or by examining the trace of their execution. Since we have recourse here to a machine, even if it is virtual, we call this technique *operational semantics.*

That strategy has a defect: it requires a non-standard machine, or more precisely, it requires a computing formalism. If, for that formalism, we used a theory already known to everybody, then we could spare ourselves that machine. A program is above all a function that transforms its input into output. "To execute" such a function does not require a complicated machine: only a few centuries of mathematical practice and culture are sufficient "to apply" it. The idea is thus to transform a program into a function (from an appropriate space of functions). We call that function its *denotation.* The remaining problem is then to understand the space of denotations.

For this purpose, λ-calculus is a good choice. Its basis is so simple that everybody agrees about how it works. The semantics of a language then becomes the process by which we associate a program with its denotation. That process is, in

fact, a function, too. The denotation of a program is the λ-term representing its meaning. Provided by these denotations, the theory of standard λ-calculus makes it possible to determine what a given program computes, whether two programs are equal, and so forth. There are, of course, many variations depending on the nature of the denotations, the type of λ-calculus chosen, the way of constructing the semantic function to associate programs and their denotations, but the structure that we've just explained is what we conventionally call *denotational semantics.*

We should also mention another approach based on the proof of programs by Floyd and Hoare. This approach is known as *axiomatic semantics.* The idea is to define each elementary form of a language by a logical formula, like this: $\{P\}form\{Q\}$. That formula indicates that if P is true before the execution of *form*, and if the execution of *form* terminates, then Q will be true. We can thus specify all the elementary forms of a language and define it axiomatically. You can clearly see the advantage of such a description for the techniques to prove correctness in that language. In contrast, this is a non-constructive procedure so it reveals nothing about the implementation of a language. It doesn't even indicate whether an implementation exists.

Even so, we have not yet exhausted the arsenal of semantics currently in use. We should also mention *natural semantics* as in [Kah87]. It favors the idea of relations (over functions) in a context derived from denotational semantics. There is also *algebraic semantics*, as in [FF89], which reasons in terms of equivalent programs by means of rewrite rules.

5.1 A Brief Review of λ-Calculus

Denotational semantics consists of defining (for a given language) a function, called the *valuation*, that associates each valid program of the language with a term in the denotation space. As the denotation space, we'll choose λ-calculus because of its structural simplicity and its proximity to Scheme, described as an efficient interpreter of λ-calculus in [SS75, Wan84].

Here we'll very briefly cover λ-calculus.[1] Its syntax is simple: the terms of λ-calculus are variables, abstractions, or applications (or combinations since in Lispian terms, functions in λ-calculus are monadic). We'll use **Variable** to indicate the set of possible, usable variables and Λ for the set of terms of λ-calculus. Λ can be defined recursively like this:

variable :	$\forall x \in \mathbf{Variable},$	$x \in \Lambda$
abstraction :	$\forall x \in \mathbf{Variable}, \forall M \in \Lambda,$	$\lambda x.M \in \Lambda$
combination	$\forall M, N \in \Lambda,$	$(MN) \in \Lambda$

As usual in Lisp, the syntax is not terribly important, and we could equally well write the terms of λ-calculus in parenthesized form.[2] The set of terms of λ-calculus is thus syntactically a subset of the terms of Scheme reduced to the sole special form `lambda`, like this:

1. There's a good introduction to λ-calculus in [Sto77, Gor88]. The "bible" of λ-calculus begins with [Bar84].
2. By the way, that's what McCarthy did in 1960 at MIT.

x `(lambda (`x`)` M`)` $(M\ \ N)$

With λ-calculus, we can write functions. There is even a rule for applying them: the *β-reduction.* It simply stipulates that when we apply a function with body M and variable x to a term N, we get a new term which is the body M of the function with the variable x replaced by the term N. We indicate that by this notation: $M[x \rightarrow N]$. That's the usual model for substitution, always used in mathematics without ever being formalized. It's also the rule used in the preceding subset of Scheme (without side effects, with `lambda` as the sole special form) when we apply a function to its argument.

$$\beta\text{-réduction :}\quad (\lambda x.M\ \ N) \xrightarrow{\beta} M[x \rightarrow N]$$

The substitution $M[x \rightarrow N]$ is a subtle operation defined by taking care not to capture unrelated variables. Such captures occur only when we make substitutions in the body of abstractions: there we again encounter the problem linked to free variables in the body of functions. The free variables of N must not be captured by the surrounding variables in M as was the case for dynamic binding. In Lispian terms, we should rather say that λ-calculus involves lexical binding. In the following definition of a substitution, we've added a few superfluous parentheses to isolate terms that are substituted.

$x[x \rightarrow N] = N$
$y[x \rightarrow N] = y$ with $x \neq y$
$(\lambda x.M)[x \rightarrow N] = \lambda x.M$
$(\lambda y.M)[x \rightarrow N] = \lambda z.(M[y \rightarrow z][x \rightarrow N])$ with $x \neq y$ and z not free in $(M\ N)$
$(M_1\ M_2)[x \rightarrow N] = (M_1[x \rightarrow N]\ M_2[x \rightarrow N])$

A *redex* is a reducible expression, or more precisely, an application in which the first term (that is, the term in the function position) is an abstraction. A β-reduction suppresses a redex. When a term contains no redex (in other words, it cannot be reduced further), we say that the term is in *normal form.* Terms in λ-calculus do not necessarily have a normal form, but when they have one, it is unique because of the Church-Rosser property.

When a term has a normal form, there exists a finite series of β-reductions that convert the original term into the normal form. An *evaluation rule* is a procedure that indicates which redex (if there is more than one) ought to be β-reduced. Unfortunately, there are both good and bad evaluation rules. One good evaluation rule is always to evaluate the redex for which the opening parenthesis is the leftmost. That rule does not necessarily minimize the computation, but it always terminates at the normal form if such a normal form exists. For that reason, we call it *normal order* or *call by name.* A bad evaluation rule—the one that Scheme follows, in fact—is known as *call by value.* In call by value, we apply the function only after having evaluated its arguments. Let's look at a few examples. Here's an example of a term without a normal form:

$(\omega\ \omega)$ with $\omega = \lambda x.(x\ x)$ since $(\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} (\omega\ \omega) \xrightarrow{\beta} \ldots$

In Scheme, that program loops and consequently leads to no term at all, thus certainly not to a normal form.

Here's a term that has a normal form, but the evaluation rule in Scheme prevents us from finding it:

$((\lambda x.\lambda y.y\ \ (\omega\omega))\ z) \xrightarrow{\beta} (\lambda y.y\ \ z) \xrightarrow{\beta} z$

In Scheme, that first argument $(\omega\ \omega)$ would be computed and loop infinitely, so the normal form would never be found.

Conversely, and also unfortunately, in Scheme we can evaluate a term without a normal form. The reason is that in Scheme we do not reduce terms in the body of `lambda` forms even if there is a redex. For example,

$\lambda x.(\omega\ \omega)$

So why do we cling to that evaluation rule in Scheme if it's not a good one? One reason is that computing by means of call by value is much more efficient than the "good rule," *call by name*, even if that one can be improved to *call by need.* Another reason is that every time Scheme achieves a value in normal form, it's the same value that would have been achieved by the "good rule" anyway since the normal form is after all unique. There is thus a frequent and fortunate coincidence between values produced by both good and bad evaluation rules.

A practical convention in λ-calculus syntactically supports functions with multiple variables by positing that $\lambda xy.M$ is the same as $\lambda x.\lambda y.M$. Reciprocally, to make it easier to apply these functions, we posit that (MN_1N_2) is in fact $((MN_1)N_2)$. That next to last example then becomes even easier to write and to understand as this simply: $(\lambda xy.y\ \ (\omega\ \omega)\ \ z)$

There you can see why it's pointless to compute the value bound to the variable x which will not even appear in the final answer.

There's great deal more we could say about the pleasures of λ-calculus, but for them, we'll direct you to fine works on the subject by [Bar84, Gor88, Dil88]. Among other things, we could enrich λ-calculus by supplementary terms, such as integers. When enlarged in that way, it's known as *applied* λ-calculus. We could also add new rules among such terms; for example, $2+2=4$ is known as a *δ-rule.* However, this kind of elaboration is not logically necessary since integers and Booleans can be encoded as λ-terms, and their arithmetic and logical operations can be as well. Even the structure of a list, with `cons`, `car`, `cdr`, can be built up that way, as in [Gor88]. **[see Ex. 4.2]**

In conclusion, we should say that λ-calculus is a highly refined theory that provides us simultaneously a simple but powerful framework for computing. In fact, β-reduction is as powerful though not so complicated as a Turing machine. Equally important, λ-calculus furnishes a basis for equality (two terms are equal if they reduce to the same third term) so we can compare terms. For all those reasons, λ-calculus is an excellent denotation space for our purposes.

5.2 Semantics of Scheme

As we've argued, then, λ-calculus is a great candidate to represent denotation. The preceding interpreter was written in a Scheme without side effects. It defined an evaluation function, `evaluate`, with the following signature:

$$\texttt{evaluate} : \textbf{Program} \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Memory} \rightarrow \textbf{Value}$$

With a slight effort, we can imagine modifying that signature by currying the first argument (the program) and thus getting this:

$$\textbf{Program} \rightarrow (\textbf{Environment} \times \textbf{Continuation} \times \textbf{Memory} \rightarrow \textbf{Value})$$

We could associate each fragment of a program with a function that expects us to provide only an environment, a continuation, and memory to indicate which value it should return. We'll stop there, that is, with the semantics of Scheme as well as the exact nature of our denotations.

valuation : **Program** → **Denotation**
Denotation : **Environment** × **Continuation** × **Memory** → **Value**

However, the people who practice denotational semantics don't really like parentheses, and they even have strong preferences about the style for elaborating denotations. First of all, they use (and abuse) Greek letters and notation shortened to the point of being elliptic and even cryptic. The reason for such habits is that after much practice, they can keep the semantics of an entire language short enough to fit on one page where anyone can see the whole of it at a glance. This advantage[3] is incompatible, of course, with long identifiers or verbose keywords. The choice of Greek letters is also motivated by the fact that denotations are written in a language that must not be confused with the language that is being defined. Since most of the languages that denotational semanticists are defining are computing languages and thus use only that limited set of characters known as ASCII, Greek letters keep things short and limit confusion. Finally, for greater security, denotations are typed, and the names of variables indicate their type.

We, too, will follow those tenacious conventions. By long and customary usage, functions are indicated by φ. Other entities are usually indicated by the Greek initial of their English name, for example, κ for continuation, α for address, ν for identifier (that is, name), π for program, σ for memory (that is, *store*). But who knows why environments are indicated by ρ?

π	**Program**	ρ	**Environment**
ν	**Identifier**	α	**Address**
σ	**Memory**	ε	**Value**
κ	**Continuation**	φ	**Function**

Each word in boldface in that chart names a *domain* representing objects handled by denotations. You see there all the objects that the preceding chapters introduced. The following chart defines those domains.

Environment = **Identifier** → **Address**
Memory = **Address** → **Value**
Value = **Function** + **Boolean** + **Integer** + **Pair** + ...
Continuation = **Value** × **Memory** → **Value**
Function = **Value*** × **Continuation** × **Memory** → **Value**

As usual, the asterisk indicates repetition. The domain **Value*** is the domain of sequences of **Value**. The sign × indicates a Cartesian product. The sign + represents the disjoint sum of domains; that is, an element of **Value** is an element of **Function** or of **Boolean** or of **Integer**, etc. All types of values are represented by a domain appearing in that sum. The disjoint sum of domains has the property that, when we take an element of **Value**, we know the exact domain that it comes from. Since we've accepted the rule that entities must be typed, we have to declare

3. This advantage is even more conspicuous if you recall that the average length of a typical scientific communication is about ten pages.

how and where these changes of domain occur. The expression ε *in***Value** means that we inject the term ϵ in the domain **Value** while $\varepsilon\,|_{\text{Integer}}$ projects the value ϵ in the domain from which it comes (assuming, of course, that ϵ is an integer).

These domains are defined recursively (a mathematically sensitive issue). Moreover, they are known as domains, rather than sets. Without going into detail, we should say that λ-calculus was developed by Alonzo Church in the thirties, but that this construction did not have a mathematical model until after work by Dana Scott around 1970. In short, λ-calculus had proved its usefulness already, but once it had a mathematical model, it was around for good. Since then, properties have been extended in several different ways, producing several different models: D^∞ or $\mathcal{P}_\omega$ in [Sco76, Sto77].

Extensionality is the property that $(\forall x, f(x) = g(x)) \Rightarrow (f = g)$. It is linked to the *$\eta$-conversion* that we often take as a supplementary rule of λ-calculus.

$$\eta\text{-conversion}: \quad \lambda x.(M\ x) \xrightarrow{\eta} M \qquad \text{with } x \text{ not free in } M$$

Strangely enough, $\mathcal{P}_\omega$ is extensional because two functions that compute the same thing at every point are equal, whereas D^∞ is not extensional. Is Mother Nature extensional?

Scott has shown that any system of domains recursively defined by means of only $\rightarrow$, $\times$, $+$, *, has a unique solution. In that sense, domains really exist.

An important principle of denotational semantics is *compositionality*: that the meaning of a fragment of a program depends only on the meaning of its components. This principle is the basis of inductive proof that we can carry out within the framework of a language defined in that way. It's also a useful principle from the point of view of the language itself: we can understand a fragment of a program independently of whatever surrounds it.

The valuation function associated with a language usually is indicated by $\mathcal{E}$. To re-enforce the distinction between a program and its semantics, we will enclose fragments of programs within semantic brackets, $[\![$ and $]\!]$. Finally, we'll present valuation case by case, that is, elementary form by elementary form.

5.2.1 References to a Variable

The simplest denotation concerns the value of a variable.

$$\mathcal{E}[\![\nu]\!] = \lambda\rho\kappa\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

The denotation of a reference to a variable (here, ν) is a λ-term that, given an environment ρ, a continuation κ, a memory σ, will determine the address associated with the variable in the environment $(\rho\ \nu)$, will submit this address to memory to get the associated value $(\sigma\ (\rho\ \nu))$, will finally return this value to the continuation accompanied by the unmodified memory (since reading memory is non-destructive).

We've used a syntactic convention to write functions with multiple arguments. Nevertheless, if we want, we can write something more exact but less legible, like this:

$$\mathcal{E}[\![\nu]\!] = \lambda\rho.\lambda\kappa.\lambda\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

No need to show off our formalism, so we will abandon this painful and obscure way of writing. In passing, to simplify things even further, we'll adopt the following

writing convention, similar to the style of **define** in Scheme. It will remove a few more layers of λ.

$\mathcal{E}[\![\nu]\!]\rho\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$

There's a possible error in the denotation of references where the variable does not appear in the environment. That erroneous situation is handled by the initial environment, ρ_0, which is:

$(\rho_0\ \nu) = \mathit{wrong}$ "**No such variable**"

When a variable is not defined in the environment, the searcher will invoke the function *wrong* which, in turn, produces a value generally indicated by $\perp$. This value is absorbent; that is, $\forall f, f\perp = \perp$. Consequently, when an erroneous situation occurs, the entire computation results in $\perp$, a clear indication that an error has occurred. In fact, it is not so much $\perp$ that has this quality of absorbency; it's more the functions that we manipulate that we call *strict*. Specifically, f is strict if and only if $f\perp = \perp$. This convention relieves us to a degree from handling errors and leaves only the most significant part of the denotation.

5.2.2 Sequence

The denotation of a sequence will use an auxiliary valuation that will be the equivalent of the function **eprogn** that you saw in earlier interpreters. We will indicate it by $\mathcal{E}^+$. We chose that name because there is necessarily at least one term in the sequence of forms that it denotes. Still conforming to that tradition, we'll indicate a succession of non-empty forms as π^+. The valuation $\mathcal{E}^+$ will convert a succession of non-empty forms π^+ into a denotation evaluating all the terms in left to right order and returning the value of the last of these forms. Its purpose is that two cases define the sequence, depending on whether it contains only one or more than one form. In Scheme, the meaning of the empty sequence (**begin**) is not specified, a point highlighted by its absence from the following cases:

$\mathcal{E}[\![(\mathbf{begin}\ \pi^+)]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$

$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$

$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$

When the sequence contains only one unique term, then the sequence is equivalent to that term. We may indicate that idea more directly, simply saying this:

$\mathcal{E}^+[\![\pi]\!] = \mathcal{E}[\![\pi]\!]$

In terms of λ-calculus, what we just wrote is an η-simplification. We'll avoid such ruffles and flourishes because they make code (or denotations) harder to read by completely masking the natural arity of functions; according to [WL93], they're often there only to look more intelligent anyway.

When the sequence contains more than one term, the denotation calculates the value of the first of these terms and forgets it in order to calculate the other terms. That's the usual definition that we see popping up once more with all the details that you've now become accustomed to. You see clearly here, in just one line, that the memory state resulting from the evaluation of the first term is the one that

serves for the evaluation of the following terms. Continuations put the evaluations in sequence, each of them passing the memory they received and used to the next one.

5.2.3 Conditional

The denotation of a conditional is highly conventional and poses hardly any difficulties if we know how to get Booleans into λ-calculus. In fact, Booleans are easy to simulate in λ-calculus. We will define the values True and False as combinators (that is, functions without free variables), like this:

$$\mathbf{T} = \lambda xy.x \qquad \text{et} \qquad \mathbf{F} = \lambda xy.y$$

Intuitively, these definitions both take two values as arguments and return the first or the second. This strategy resembles that of the logical connector **If** which corresponds to the equations:

$$\mathbf{If}(true, p, q) = p \qquad \text{and} \qquad \mathbf{If}(false, p, q) = q$$

Careful: this logical connector has nothing to do with the special form `if` in Scheme. Here, we're not talking about the order of evaluation, but only about the fact that **If** is a *function* which could be defined by a truth table but which is written more simply like this:

$$\mathbf{If}(c, p, q) = (\neg c \vee p) \wedge (c \vee q)$$

If we take into account the code we choose for Booleans, there is a simple way to simulate **If** in λ-calculus: we write this new combinator **IF**, like this:

$$\mathbf{IF}\ c\ p\ q = (c\ p\ q)$$

Just as in ordinary logic, this definition says nothing about the order of the computation but states only a relation among three values. Seen as a function, **If** returns its second argument if the first value is True; it returns the third argument otherwise. Like [FW84], we'll call this function `ef`. In more Lispian terms, we can approximate this function like this:

```
(define (ef v v1 v2)
  (v v1 v2) )
```

To make the notation for a conditional more legible, for the moment, we'll adopt the following syntax from [Sch86]:

$\epsilon_0 \rightarrow \epsilon_1 [] \epsilon_2$

As for R^{4}RS, it uses this:

$\varepsilon_1 \rightarrow \varepsilon_2, \varepsilon_3$.

With these ideas in mind, we articulate the denotation of a conditional like this:

$$\begin{array}{l}\mathcal{E}[\![(\texttt{if}\ \pi\ \pi_1\ \pi_2)]\!]\rho\kappa\sigma = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.\ (\mathit{boolify}\ \varepsilon) \\ \qquad\qquad \rightarrow\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma_1) \\ \qquad\qquad []\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1)\ \ \sigma)\end{array}$$

The function *boolify* converts a value into a Boolean since every value in Scheme is implicitly a truth value. A conditional thus starts by evaluating its condition with a new continuation that decides which branch of the conditional to follow according to the value of the condition. Careful: the conditional in λ-calculus looks as though it behaves like our old friend `if-then-else`, but they differ in a significant respect: nothing, absolutely nothing, indicates the order of evaluation. Inside λ-calculus, we could very well evaluate the condition and two branches of a conditional in parallel in order to choose among them once all the computations are complete.

The consequences of that fact are important because we cannot look at λ-calculus (in some ways, the language in which we are defining a new interpreter) as we used to look at Scheme: there is no idea in λ-calculus of any order in evaluation. To articulate the denotation defining a conditional, we could say that the value of a conditional is (among the two possible values) the one indicated by the value of the condition.

There is no unfair competition nor hidden side effects if we evaluate the two branches of a conditional in parallel since each has its own set of parameters defining the computation, and besides, we're in a pure language completely stripped of side effects. For example, there's no problem with the following expression, even though you might think that if we don't evaluate the components in the "right" order, we're courting disaster:

```
(if (= 0 q) 1 (/ p q))
```

That expression tests whether the divisor is null before the division, and in that case, it returns the value 1. Its denotation simply states that the choice between the value 1 and the quotient of `p` divided by `q` will be made according to whether or not `q` is null.

The difficulty of this operator comes from the habit we might have acquired from Scheme and from the fact that we see λ-calculus through the evaluation rule for Scheme. To a degree, we diminish the distance between Scheme and λ-calculus by rewriting the denotation of a conditional like this:

$$\mathcal{E}[\![\texttt{(if } \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.((\mathit{boolify}\ \varepsilon) \rightarrow\ \mathcal{E}[\![\pi_1]\!][]\ \mathcal{E}[\![\pi_2]\!]\rho\ \kappa\ \sigma_1)\ \sigma)$$

We've introduced a little sequentiality here since the redices have been ordered. The value of the conditional serves only to choose the denotation of the elected branch, the one invoked independently.

The problem with this new definition is knowing whether it is still equivalent to the preceding one. It entails proving whether the two denotations are the same for a given program. To do so, it suffices to show this:

$$(\mathit{boolify}\ \varepsilon) \rightarrow\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma)[]\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma) = ((\mathit{boolify}\ \varepsilon) \rightarrow\ \mathcal{E}[\![\pi_1]\!][]\ \mathcal{E}[\![\pi_2]\!]\rho\ \kappa\ \sigma)$$

That equivalence is obviously false in Scheme because of the fact that evaluation is ordered. To convince ourselves, all we have to do is make π_2 an expression that loops. However, denotations are terms from λ-calculus and thus have to be compared according to the laws of λ-calculus. Consequently, we simply distribute the application of a conditional.

When we see how cryptic that syntax is, we're prompted to adopt a more eloquent notation for denotational `if-then-else`:

$\mathcal{E}[\![(\texttt{if}\ \pi\ \pi_1\ \pi_2)]\!]\rho\kappa\sigma =$
$(\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.($ **if** $(boolify\ \varepsilon)$
then $\mathcal{E}[\![\pi_1]\!]$
else $\mathcal{E}[\![\pi_2]\!]$
endif $\rho\ \kappa\ \sigma_1)\ \sigma)$

5.2.4 Assignment

The denotation for assignment is simple. The version we present here has the newly assigned value for its value.

$\mathcal{E}[\![(\texttt{set!}\quad \nu\ \pi)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$

$f[y \rightarrow z] = \lambda x.$ **if** $y = x$ **then** z **else** $(f\ x)$ **endif**

Memory is extended to reflect the assignment that's carried out, so the value ε is associated with the address of the variable ν. We produce this extension by using suggestive *ad hoc* notation: $\sigma[\alpha \rightarrow \varepsilon]$. There is other, similar notation, like $[\alpha \rightarrow \varepsilon]\sigma$ in [Sch86] or $[\varepsilon/\alpha]\sigma$ in [Sto77] or even $\sigma[\varepsilon/\alpha]$ in [Gor88, CR91b].

Let's enrich our λ-calculus with a few supplementary functions. We'll write sequences between these delimiters: $\langle$ and $\rangle$. We'll indicate the concatenation of sequences by this sign: §. To extract terms from a sequence, we'll indicate the extraction of the i^{th} term of a sequence by $\langle\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n\rangle \downarrow i$. To indicate the truncation of the first i terms from a sequence, (that is, to get this: $\langle\varepsilon_{i+1}, \ldots, \varepsilon_n\rangle$), we'll write: $\langle\varepsilon_1, \ldots, \varepsilon_n\rangle \dagger i$. The notation $\#\varepsilon^*$ indicates the length of the sequence ε^*. All this notation can be defined in pure λ-calculus, but doing so obscures the presentation a bit, so without sacrilege, we'll use $\downarrow_1$, $\dagger 1$, $\#$, and § as the denotational equivalent of **car**, **cdr**, **length**, and **append**.

We'll extend the extension of the environment itself to a group of points and images. In what follows, we'll assume that the two sequences, x^* and y^*, have the same length.

$f[y^* \xrightarrow{*} z^*] =$ **if** $\#y^* > 0$ **then** $f[y^* \dagger 1 \xrightarrow{*} z^* \dagger 1][y^* \downarrow_1 \rightarrow z^* \downarrow_1]$ **else** f **endif**

5.2.5 Abstraction

As a first effort, we'll denote only functions with fixed arity, that is, those lacking a dotted variable.

$\mathcal{E}[\![(\texttt{lambda}\ (\nu^*)\ \pi^+)]\!]\rho\kappa\sigma =$
$(\kappa\ in\textbf{Value}(\lambda\ \varepsilon^*\kappa_1\sigma_1.$
if $\#\varepsilon^* = \#\nu^*$
then $allocate\ \sigma_1\ \#\nu^*$
$\lambda\ \sigma_2\alpha^*.$
$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])$
else *wrong* "**Incorrect arity**"
endif $)\ \sigma)$

The injection *in***Value** takes a λ-term representing a function and converts it to a value. The inverse operation will appear in the functional application.

When a function is invoked and after its arity has been verified, new addresses are allocated to be associated with variables of the function and to contain the values that they take for this invocation. Allocation of addresses in memory is carried out by the function *allocate*; it takes memory, the addresses to allocate, and a kind of "continuation" that it will invoke on the allocated addresses and the new memory where these addresses have been allocated. *allocate* is a real function, so when its arguments are equal, it makes corresponding equal results, but its exact definition is generally left vague in order not to overburden denotations. Besides, its definition is so low-level technically that it is not very interesting.

The function *allocate* is polymorphic; α can represent any type. Here's it signature:

$$\mathbf{Memory} \times \mathbf{NaturalInteger} \times (\mathbf{Memory} \times \mathbf{Address}^* \to \alpha) \to \alpha$$

(A precise definition of *allocate* appeared in the previous chapter. **[see p. 132]**)

5.2.6 Functional Application

Functions are meant to be applied, so here's the denotation of application. Once again, we'll use a new auxiliary valuation: $\mathcal{E}^*$, a kind of denotational `evlis`.

$$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\text{Function}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

Continuations that use $\mathcal{E}^*$ don't wait for a value but for a sequence of values. That was not the case for the valuation $\mathcal{E}^+$. The values of κ in these definitions thus have the following type:

$$\mathbf{Value}^* \times \mathbf{Memory} \to \mathbf{Value}$$

5.2.7 `call/cc`

Our fast trip through denotations would not be complete without a definition of a function essential to the semantics of Scheme: `call/cc`. We define it like this:

$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$
*in*Value$(\lambda\ \varepsilon^*\kappa\sigma.$
if $1 = \#\varepsilon^*$
then $(\ \varepsilon^*\downarrow_1|_{\text{Function}}$
$\langle$*in*Value$(\lambda\ \varepsilon^*{}_1\kappa_1\sigma_1.$
if $1 = \#\varepsilon^*{}_1$
then $(\kappa\ \varepsilon^*{}_1\downarrow_1\ \sigma_1)$
else *wrong* "`Incorrect arity`"
endif $)\rangle\ \kappa\ \sigma)$
else *wrong* "`Incorrect arity`"
endif $)$

Notice that there are the successive injections and projections between the various domains of **Value** and **Function**. The denotation itself is not really any more complicated than the other ways of programming `call/cc` that you've already seen in Chapter 3. **[see p. 95]**

5.2.8 Tentative Conclusions

We've just managed, case by case, to define a function that associates a λ-term with every program. There's no doubt about the existence of this function because we have used the principle of composition to construct it. If we allow syntactically recursive programs as in [Que92a], then we need a little more theory to prove that we have a well defined function.

For now, we can show that the semantics of primitive special forms in Scheme can be seen at a glance: Table 5.1.

Of course, the special form `quote` is missing (quotation poses problems, as you remember [see p. 140]), and we won't see functions with variable arity until later. We're also missing `eq?` for comparing functions and a number of other predefined functions. We have, however, gotten `call/cc`, appearing in its simplest guise. Other functions of the predefined library, like `cons`, `car`, `set-cdr!` can be simulated in this subset by closures without adding any hidden features. On that basis, we can talk about the essentials independently of any additional functions that we might add. The basis of the language, that is, its special forms and primitive functions like `call/cc`, are enough to anchor our understanding.

Still, we insist that being able to see the essentials of Scheme in a single table with this degree of detail is well worth such austere coding practices. In this way, we're bringing to an end our progress since taking off from using an entire Scheme to define an approximate Scheme in the first chapter. Now we've arrived at using λ-calculus to define an entire Scheme.

5.3 Semantics of λ-calcul

Defining the essentials of a language in so few signs is one of the attractions of Scheme. Indeed, for just that reason, functional languages generally become veritable experimental linguistic laboratories for introducing new constructions and for studying them in terms of basic, fundamental, and common traits. Depending on the characteristics that we want to analyze, we could, of course, start from a more restricted linguistic basis, such as λ-calculus itself. That assertion is not really tautologic; it provides a second example of the denotation of a language that is different but related: the semantics of λ-calculus itself.

Let's focus first on the syntax. Since syntax has no importance for us here other than clarifying our ideas, we'll deliberately choose a Scheme-like syntax:

x `(lambda (`x`)` M`)` `(`M N`)`

Now let's determine the domains to manipulate. λ-calculus has no idea of assignment nor continuation, and we'll take advantage of those facts. While we're at it, we'll limit ourselves to a non-applied λ-calculus with closures as the only values. Here, then, are the domains. You can see that they are a restricted set of the domains of Scheme.

$$\mathcal{E}[\![\nu]\!] = \lambda\rho\kappa\sigma.(\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\quad \nu\ \pi\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$$

$$\begin{array}{l}\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\kappa\sigma = \\ (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.\ \textbf{if}\ (\textit{boolify}\ \varepsilon) \\ \qquad\qquad \textbf{then}\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \kappa\ \sigma_1) \\ \qquad\qquad \textbf{else}\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \kappa\ \sigma_1) \\ \qquad\qquad \textbf{endif}\quad \sigma)\end{array}$$

$$\begin{array}{l}\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\ (\kappa\ \textit{in}\textbf{Value}(\lambda\ \varepsilon^*\kappa_1\sigma_1. \\ \qquad\qquad \textbf{if}\ \#\varepsilon^* = \#\nu^* \\ \qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ \#\nu^* \\ \qquad\qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\ \qquad\qquad\qquad\qquad\quad (\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\ \qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad \textbf{endif}\quad)\ \sigma)\end{array}$$

$$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\textbf{Function}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \kappa\ \sigma_1)\ \sigma)$$

$$\begin{array}{l}(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) = \\ \textit{in}\textbf{Value}(\lambda\ \varepsilon^*\kappa\sigma. \\ \qquad\qquad \textbf{if}\ 1 = \#\varepsilon^* \\ \qquad\qquad \textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Function}} \\ \qquad\qquad\qquad\qquad \langle\textit{in}\textbf{Value}(\lambda\ \varepsilon^*{}_1\kappa_1\sigma_1. \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if}\ 1 = \#\varepsilon^*{}_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \sigma_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif}\quad)\rangle\ \kappa\ \sigma) \\ \qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\ \qquad\qquad \textbf{endif}\quad)\end{array}$$

Table 5.1 Essential Scheme

π	**Program**		
ν	**Identifier**		
ρ	**Environment**	=	**Identifier** $\rightarrow$ **Value**
ε	**Value**	=	**Function**
φ	**Function**	=	**Value** $\rightarrow$ **Value**

We'll call the valuation function $\mathcal{L}$. It will associate a λ-term with a denotation, that is, another λ-term. Consequently, the valuation has this signature:

$$\mathcal{L} : \textbf{Program} \rightarrow (\textbf{Environment} \rightarrow \textbf{Value})$$

The only task left to do is to define that function, case by case, by analyzing the various syntactic possibilities, as in Table 5.2.

$$\mathcal{L}[\![\nu]\!]\rho = (\rho\ \nu)$$

$$\mathcal{L}[\![\texttt{(lambda (}\nu\texttt{) }\pi\texttt{)}]\!]\rho = \lambda\varepsilon.(\mathcal{L}[\![\pi]\!]\ \rho[\nu \rightarrow \varepsilon])$$

$$\mathcal{L}[\![(\pi\ \pi')]\!]\rho = ((\mathcal{L}[\![\pi]\!]\ \rho)(\mathcal{L}[\![\pi']\!]\ \rho))$$

Table 5.2 Semantics of λ-calculus

This denotation is scrupulous with respect to the order of evaluation of terms in a functional application (or combination). In effect, the combination is transformed into a combination and nothing is said about the order.

The valuation $\mathcal{L}$ is defined recursively. There's no problem in doing that here because of compositionality: all the recursive calls are carried out in smaller programs, and the terminal case is provided by reference to the variables.

λ-calculus is a special case for us because we already have a very clear idea of the semantics of its terms. We can prove (see [Sto77, page 158]) that the change to denotations preserves all its necessary properties and, notably, β-reduction.

Denotation from λ-calculus is the basis of denotation in functional languages with no side effects nor continuations. When side effects and continuations are introduced, it's generally necessary to introduce an explicit order of evaluation to handle them correctly. If we also want to add assignment, then we have to split the environment by introducing addresses, the famous boxes of Chapter 4 or references as in ML. [see p. 114]

5.4 Functions with Variable Arity

In this section, we'll show how to incorporate functions with variable arity into Scheme. These functions are special in that they handle excess arguments that they receive as a list. Consequently, at every invocation, there is an allocation disguised as dotted pairs, which, by the way, can be very expensive if these functions are used frequently. In a certain way, the antidote to functions with a list of dotted variables is the primitive function `apply`; it converts a list of values into the missing

arguments. Functions of variable arity are thus inevitably associated in Scheme with lists, so we must define the denotations of the usual functions (like `cons`, `car`, `set-cdr!`) on lists.

Dotted pairs are represented by the domain **Pair**. It appears as one of the components of the disjoint sum defining **Value**. The domain **Pair** itself will be defined as in the preceding chapter, by the Cartesian product of two addresses.

$$\begin{aligned} \textbf{Value} &= \textbf{Function} + \textbf{Boolean} + \textbf{Integer} + \textbf{Pair} + \ldots \\ \textbf{Pair} &= \textbf{Address} \times \textbf{Address} \end{aligned}$$

The denotations of `cons`, `car`, and `set-cdr!` (we need to show at least one side effect on dotted pairs) are highly conventional; they hardly differ at all from the programming style we used in the preceding chapter. The only difficulties now are in notation because, for example, we have to interpret $\varepsilon^* \downarrow_1 |_{\mathbf{Pair}} \downarrow_2$ in this way: $\varepsilon^* \downarrow_1$ is the first argument of the function, a value, which is then projected on the domain of dotted pairs, $|_{\mathbf{Pair}}$; if it is not a dotted pair, we get $\perp$; finally, we extract the address of its `cdr` from this dotted pair, that is, its second component, $\downarrow_2$.

$$\begin{aligned} &(\sigma_0(\rho_0[\![\texttt{cons}]\!])) = \\ &in\mathbf{Value}(\lambda\varepsilon^*\kappa\sigma.\ \textbf{if } 2 = \#\varepsilon^* \\ &\qquad \textbf{then } \mathit{allocate}\ \sigma\ 2\ \ \lambda\sigma_1\alpha^*.(\kappa\ in\mathbf{Value}(\langle\alpha^*\downarrow_1, \alpha^*\downarrow_2\rangle)\ \sigma_1[\alpha^* \xrightarrow{*} \varepsilon^*]) \\ &\qquad \textbf{else } \mathit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\ &\qquad \textbf{endif }) \end{aligned}$$

$$\begin{aligned} &(\sigma_0(\rho_0[\![\texttt{car}]\!])) = \\ &in\mathbf{Value}(\lambda\varepsilon^*\kappa\sigma.\ \textbf{if } 1 = \#\varepsilon^* \\ &\qquad \textbf{then } (\kappa\ (\sigma\ \varepsilon^*\downarrow_1|_{\mathbf{Pair}}\downarrow_1)\ \sigma) \\ &\qquad \textbf{else } \mathit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\ &\qquad \textbf{endif }) \end{aligned}$$

$$\begin{aligned} &(\sigma_0(\rho_0[\![\texttt{set-cdr!}]\!])) = \\ &in\mathbf{Value}(\lambda\varepsilon^*\kappa\sigma.\ \textbf{if } 2 = \#\varepsilon^* \\ &\qquad \textbf{then } (\kappa\ \varepsilon^*\downarrow_1\ \ \sigma[\varepsilon^*\downarrow_1|_{\mathbf{Pair}}\downarrow_2 \rightarrow \varepsilon^*\downarrow_2]) \\ &\qquad \textbf{else } \mathit{wrong}\ \text{“}\texttt{incorrect arity}\text{”} \\ &\qquad \textbf{endif }) \end{aligned}$$

Once the structure of lists is known, `apply` is simple to specify. We gather the arguments from the second (since the first argument is the function to invoke) to the last, excluded. This gathering must be a list that we flatten. We then gather the successive terms into a sequence of arguments to which we apply the specified function.

$$\begin{aligned} &(\sigma_0(\rho_0[\![\texttt{apply}]\!])) = \\ &in\mathbf{Value}(\lambda\varepsilon^*\kappa\sigma.\ \textbf{if } \#\varepsilon^* \geq 2 \\ &\qquad \textbf{then } (\varepsilon^*\downarrow_1|_{\mathbf{Function}}\ (\mathit{collect}\ \varepsilon^* \dagger 1)\ \kappa\ \sigma) \\ &\qquad\qquad \textbf{whererec } \mathit{collect} = \lambda\varepsilon^*{}_1.\ \textbf{if } \varepsilon^*{}_1 \dagger 1 = \langle\rangle \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } (\mathit{flat}\ \varepsilon^*{}_1\downarrow_1) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \langle\varepsilon^*{}_1\downarrow_1\rangle\S(\mathit{collect}\ \varepsilon^*{}_1 \dagger 1) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\ &\qquad\qquad \textbf{and } \mathit{flat} = \lambda\varepsilon.\ \textbf{if } \varepsilon \in \mathbf{Pair} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \langle(\sigma\ \varepsilon\ |_{\mathbf{Pair}}\downarrow_1)\rangle\S(\mathit{flat}\ (\sigma\ \varepsilon\ |_{\mathbf{Pair}}\downarrow_2)) \end{aligned}$$

```
                    else ⟨⟩
                    endif
          else wrong "Incorrect arity"
          endif )
```

Now we can actually get to functions with variable arity. For them, we'll change the denotation of the special form `lambda` and introduce a particular valuation function. Its only role will be to create bindings between variables and values. We'll call this new valuation $\mathcal{B}$ for *binding*. Its signature is related to the signature of denotations, the signature of $\mathcal{E}$. More precisely, we'll use an abbreviation τ as a shortcut and write this:

$$\begin{aligned} \tau &\equiv \mathbf{Value}^* \times \mathbf{Environment} \times \mathbf{Continuation} \times \mathbf{Memory} \\ \mathcal{E}: &\ \mathbf{Program} \to \tau \to \mathbf{Value} \\ \mathcal{B}: &\ \mathbf{ListofVariables} \to (\tau \to \mathbf{Value}) \times \tau \to \mathbf{Value} \end{aligned}$$

$\mathcal{B}$, the binding valuation, binds a variable to the address that contains its value only after the arity has been verified by `lambda`. When that succeeds, it runs through the list of variables, and the corresponding locations are allocated one by one. The lexical environment where the function was defined is progressively enlarged. Finally, the body of the function is evaluated. Here, then, is the new way of presenting functions of fixed arity:

$$\begin{aligned} &\mathcal{E}[\![\texttt{(lambda } (\nu^*)\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\ &(\kappa\ in\mathbf{Value}(\lambda\varepsilon^*\kappa_1\sigma_1.\ \mathbf{if}\ \#\varepsilon^* = \#\nu^* \\ &\qquad\qquad \mathbf{then}\ ((\mathcal{B}[\![\nu^*]\!]\ \lambda\varepsilon^*{}_1\rho_1\kappa_2\sigma_2.(\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2))\varepsilon^*\ \rho\ \kappa_1\ \sigma_1) \\ &\qquad\qquad \mathbf{else}\ wrong\ \text{“}\texttt{Incorrect arity}\text{”} \\ &\qquad\qquad \mathbf{endif}\ )\ \sigma) \end{aligned}$$

$$\mathcal{B}[\![\nu\ \nu^*]\!]\mu = (\mathcal{B}[\![\nu]\!]\ (\mathcal{B}[\![\nu^*]\!]\ \mu))$$

$$\mathcal{B}[\![\ ]\!]\mu = \mu$$

$$\begin{aligned} &\mathcal{B}[\![\nu]\!]\mu = \\ &\lambda\varepsilon^*\rho\kappa\sigma.allocate\ \sigma\ 1\ \ \lambda\sigma_1\alpha^*.\ \mathbf{let}\ \alpha = \alpha^*\downarrow_1 \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (\mu\ \varepsilon^*\dagger 1\ \rho[\nu \to \alpha]\ \kappa\ \sigma_1[\alpha \to \varepsilon^*\downarrow_1]) \end{aligned}$$

To handle functions of variable arity, we will introduce a new case in the denotation of `lambda` with a dotted list of variables as well as the appropriate binding clause. That clause takes a sequence of values, converts it into a list of values (a real list made up of real dotted pairs) and binds it to the dotted variable. The co-existence of functions with multiple arity, of the function `apply`, and of side effects (as specified in Scheme) means that fresh dotted pairs have to be allocated. So the following expression should return False:

```
(let ((arguments (list 1 2 3)))
  (apply (lambda args (eq? args arguments)) arguments) )
```

An evaluator that wants to share dotted pairs has to prove beforehand that doing so does not alter the semantics of the program.

Here, finally, are the denotations of functions with multiple arity:

$$\begin{aligned} &\mathcal{E}[\![\texttt{(lambda } (\nu^*\ .\ \nu)\ \pi^+\texttt{)}]\!]\rho\kappa\sigma = \\ &(\kappa\ in\mathbf{Value}(\lambda\ \varepsilon^*\kappa_1\sigma_1. \end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\quad \textbf{if } \#\varepsilon^* \geq \#\nu^* \\
\qquad\qquad\quad \textbf{then } (\ (\mathcal{B}[\![\nu^*]\!]\ (\mathcal{B}[\![\ .\quad \nu]\!]\ \lambda\ \varepsilon^*{}_1\rho_1\kappa_2\sigma_2. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathcal{E}^+[\![\pi^+]\!]\ \rho_1\ \kappa_2\ \sigma_2)\ ))) \\
\qquad\qquad\qquad\quad \varepsilon^*\ \rho\ \kappa_1\ \sigma_1) \\
\qquad\qquad\quad \textbf{else } \textit{wrong}\ \text{“Incorrect arity”} \\
\qquad\qquad\quad \textbf{endif}\quad)\ \sigma)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}[\![\ .\quad \nu]\!]\mu = \\
\lambda\ \varepsilon^*\rho\kappa\sigma. \\
\quad (\textit{listify}\ \varepsilon^*\ \sigma\ \lambda\ \varepsilon\sigma_1. \\
\qquad\qquad\quad \textit{allocate}\ \sigma_1\ 1 \\
\qquad\qquad\quad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\quad \textbf{let}\ \alpha = \alpha^*\downarrow_1 \\
\qquad\qquad\qquad\quad \textbf{in}\ (\mu\ \langle\rangle\ \rho[\nu \rightarrow \alpha]\ \kappa\ \sigma_2[\alpha \rightarrow \varepsilon])\quad) \\
\quad \textbf{whererec}\ \textit{listify} = \lambda\ \varepsilon^*{}_1\sigma_1\kappa_1. \\
\qquad\qquad\qquad\quad \textbf{if } \#\varepsilon^*{}_1 > 0 \\
\qquad\qquad\qquad\quad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ 2 \\
\qquad\qquad\qquad\qquad\quad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ \kappa_2 = \lambda\ \varepsilon\sigma_3. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\kappa_1\ \textit{in}\textbf{Value}(\alpha^*)\ \sigma_3[\alpha^*\downarrow_2 \rightarrow \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\textit{listify}\ \varepsilon^*{}_1 \dagger 1\ \sigma_2[\alpha^*\downarrow_1 \rightarrow \varepsilon^*{}_1\downarrow_1]\ \kappa_2) \\
\qquad\qquad\qquad\quad \textbf{else}\ (\kappa_1\ \textit{in}\textbf{Value}(\langle\rangle)\ \sigma_1) \\
\qquad\qquad\qquad\quad \textbf{endif}
\end{array}
$$

Here you can see that the denotation of non-trivial characteristics of Scheme, such as functions of variable arity, necessitates a non-negligible denotational programming effort. In fact, we've written a veritable interpreter. When you compare the elegance of the description of the kernel with the preceding lines, you see that adding an interesting but minor trait nearly doubles the size of the definition; we won't mention how cryptic the addition makes the definition.

5.5 Evaluation Order for Applications

Occasionally in electronic news groups, there are violent, almost religious flames about this characteristic in the various standards for Scheme. Up to now, this point has been unchanging: that the order for evaluating terms of a functional application is not specified. Not specifying the evaluation order discourages everyone from writing programs that depend on evaluation order, but it also makes searching for errors in this area particularly difficult. A great many programs, even some written by recognized authorities, depend obscurely on the order of evaluation, especially when continuations are mixed in. For our part, we favor left to right order; it corresponds to the conventional direction for reading many languages, and it makes searching for errors at least more systematic.

Two objections from the opposing party are interesting in this context. The first is that many languages do not impose this order. Among them, the C programming language does not. Thus if we directly compile `(foo (f x) (g x y))` in C as `foo(f(x),g(x,y))`, then nothing is sure about the order produced, and it can be expensive to impose order here.

The second argument is more subtle. In a world without order, we can impose one by using **begin** to impose a sequence explicitly. In contrast, if an order is imposed, then there is no longer a way to write a program where the order is left unspecified. We're reduced in such a case to something like using a random order generator that prescribes a particular order to follow at execution time—an expensive solution at best. [see **Ex. 5.4**]

Order is not imposed in C so that the compiler can consider the terms of an application in whatever order it needs or finds most efficient, notably with respect to allocating registers.

Explicit order simplifies program debugging by eliminating one source of indeterminism. If the order is not prescribed, two executions of the same program can turn out differently, not leading to the same result. Consider this example[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 ) )
```

The internal function **amb** returns True or False according to the evaluation order. If the order changes dynamically, then the function **dynamically-changing-evaluation-order?** halts; otherwise, it loops indefinitely. R^{4}RS does not stipulate anything that would necessarily make this program loop or halt. If order were imposed, then obviously this program loops forever.

In this discussion, we have to distinguish the implementation language clearly. An implementation absolutely must choose an evaluation order when it evaluates an application. It might decide to adopt left to right order for all applications, or right to left, like MacScheme, or some other order, depending on its whim of the moment. I know of no implementation that, having chosen an order for a particular application, changes that order dynamically. In practice, order is usually chosen at compile time and it's not questioned afterwards. The language may not impose an order, but the implementation is free to choose one and publish the choice; that's legal.

The problem that interests us now is how to specify that no order is prescribed. The solution we propose is to change the structure of denotation subtly. A denotation has been a λ-term waiting for an environment, a continuation, and memory to return a value. Our choice about returning a value has been somewhat limiting because in fact the result of an evaluation is twofold: it includes a value and, in addition, the resulting state of memory. For that reason, we could equally well take the pair (value, memory) as the image of a denotation. We'll actually transcend this question altogether by naming a codomain of denotations, **Result**, and we'll leave its definition a little vague for the moment.

Since more than one evaluation order is possible, then rather than returning a unique response, a denotation could return a set of possible responses among which one would be chosen according to obscure criteria left to the discretion of the implementation. We'll modify the preceding valuation $\mathcal{E}$ to return the set of all

4. This program was implemented in collaboration with Matthias Felleisen.

possible responses now, and we'll introduce $\mathcal{N}$ to choose one among them. We'll indicate the set of parts of Q as $\mathcal{P}(Q)$. The signatures then become these:

$$\begin{aligned}\mathcal{E}: \quad & \textbf{Program} \to \textbf{Value}^* \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Memory} \\ & \qquad \to \mathcal{P}(\textbf{Result}) \\ \mathcal{N}: \quad & \textbf{Program} \to \textbf{Value}^* \times \textbf{Environment} \times \textbf{Continuation} \times \textbf{Memory} \\ & \qquad \to \textbf{Result}\end{aligned}$$

The valuation $\mathcal{N}$ is defined straightforwardly as calling the function *oneof*; the definition of *oneof* is left to the implementation; it chooses the one it wants among all the possible results.

$$\mathcal{N}[\![\pi]\!]\rho\kappa\sigma = (oneof\ (\mathcal{E}[\![\pi]\!]\ \rho\ \kappa\ \sigma))$$

Now we have to modify the denotation of a functional application to return all possible values. Semantics inspires the technique we'll use. To say that there is no evaluation order is to say that when confronted with an application, the evaluator chooses one of the terms, say, π_{i_0}, evaluates it to get its value, ε_{i_0}, then chooses a second term, say, π_{i_1}, evaluates it in turn and gets ε_{i_1}, and continues that way to the last term. The values $\varepsilon_{i_0}, \varepsilon_{i_1}, \ldots$ are then re-ordered into $\varepsilon_0, \varepsilon_1 \ldots$, and the first among them is applied to the others. Notice the order of choices as it's been described. It would be quite different to fix the order of evaluation of all terms before the evaluation of the first one, as was done in R[3,4]RS. Let's take an example. Not only will this function print an undetermined digit when called, but the returned continuation when invoked will also print another undetermined digit.

```
(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)) ) )) )
```

The denotation of the application without order will "implement" exactly what we articulated earlier. It will consider all the possible choices of terms, aided by the function *forall* which applies its first argument (a ternary function) to all the possible cuts of its second argument (a list). The function *cut* chops a list into two segments; the first contains the first i terms of the list; all the other terms occur in the second. The continuation (the third argument of *cut*) is finally applied to these two segments. The programming is quite subtle, a good example of the continuation passing style. Sophie Anglade and Jean-Jacques Lacrampe, in [ALQ95], collaborated on the following definitions.

$$\begin{aligned}&\mathcal{E}[\![(\pi_0\ \pi_1\ \ldots\ \pi_n)]\!]\rho\kappa\sigma = \\ &((possible\text{-}paths\ \langle\mathcal{E}[\![\pi_0]\!], \mathcal{E}[\![\pi_1]\!], \ldots \mathcal{E}[\![\pi_n]\!]\rangle)\rho\ \lambda\varepsilon^*\sigma_1.(\varepsilon^* \downarrow_1|_{\textsc{Function}}\ \varepsilon^* \dagger 1\ \kappa\ \sigma_1)\ \sigma)\end{aligned}$$

$$\begin{aligned}&(possible\text{-}paths\ \mu^+) = \\ &\lambda\ \rho\kappa\sigma. \\ &\quad \textbf{if}\ \#\mu^+ \dagger 1 > 0 \\ &\quad \textbf{then}\ (forall\ \lambda\ {\mu^+}_1\mu{\mu^+}_2. \\ &\qquad\qquad (\mu\ \rho\ \lambda\ \varepsilon\sigma_1. \\ &\qquad\qquad\qquad (\ (possible\text{-}paths\ {\mu^+}_1 \S {\mu^+}_2) \\ &\qquad\qquad\qquad\ \rho\ \lambda\ \varepsilon^*\sigma_2. \\ &\qquad\qquad\qquad\qquad \textbf{let}\ \kappa_1 = \lambda\ {\varepsilon^*}_1{\varepsilon^*}_2.\end{aligned}$$

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\kappa\ \varepsilon^*{}_1 \S \langle\varepsilon\rangle \S \varepsilon^*{}_2\ \sigma_2) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\,(cut\ \#\mu^+{}_1\ \varepsilon^*\ \kappa_1)\ \ \sigma_1)\ \ \sigma)\ \ \mu^+) \\ \textbf{else}\ (\ \mu^+ \downarrow_1 \\ \qquad \rho\ \lambda\ \varepsilon\sigma_1. \\ \qquad\quad (\kappa\ \langle\varepsilon\rangle\ \sigma_1)\ \ \sigma) \\ \textbf{endif} \end{array}$$

$$\begin{array}{l} (forall\ \varphi\ l) = \\ (loop\ \langle\rangle\ l\downarrow_1\ \ l \dagger 1) \\ \textbf{whererec}\ loop = \lambda l_1 \varepsilon l_2.(\varphi\ l_1\ \varepsilon\ l_2) \cup\ \textbf{if}\ \#l_2 > 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (loop\ l_1 \S \langle\varepsilon\rangle\ l_2 \downarrow_1\ \ l_2 \dagger 1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \emptyset \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif} \end{array}$$

$$\begin{array}{l} (cut\ \iota\ \varepsilon^*\ \kappa) = \\ (accumulate\ \langle\rangle\ \iota\ \varepsilon^*) \\ \textbf{whererec}\ accumulate = \lambda l \iota_1 l_1.\ \ \textbf{if}\ \iota_1 > 0 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (accumulate\ \langle l_1 \downarrow_1\rangle \S l\ \iota_1 - 1\ l_1 \dagger 1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\kappa\ (reverse\ l)\ l_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{endif} \end{array}$$

For all these variations about the order of evaluation, all were sequential—a point imposed by Scheme. The terms might not be evaluated in a particular order (they are "disordered," as it were), but they still have to be evaluated one after another. In that light, the only possible responses to the following program are `(3 5)` or `(4 3)`, but in no case is `(3 3)` possible.

```
(let ((x 1)(y 2))
  (list (begin (set! x (+ x y)) x)
        (begin (set! y (+ x y)) y) ) )
```

In contrast, the new valuation $\mathcal{E}$ applied to the following program allows two possible responses: 1 or 2. A given implementation will compute only one, but it will be one of those foreseen.

```
(call/cc (lambda (k) ((k 1) (k 2))))   → 1 or 2
```

5.6 Dynamic Binding

The idea of dynamic binding is not only important but also useful, and it has prevailed so long in Lisp interpreters that we really have to give it one of its possible denotations. To do so, we'll extend the denotation of the Scheme we've already explained so far, and we'll add to that a few special forms for handling this new type of binding. There are, in fact, many kinds of dynamic binding using special forms or specialized functions. [see p. 50] Traditionally, Scheme exploits this latter solution to avoid tampering with the denotation of its kernel. Unfortunately, certain functions, though they respect the function calling protocol, perturb this ideal. Just think about `call/cc`: it requires continuations. The denotation of dynamic binding implies the existence of an environment for storing these bindings. For that reason, we'll introduce a new environment wherever needed. The kernel of a Scheme without embellishments appears in Table 5.3.

$$\mathcal{E}[\![\nu]\!]\rho\delta\kappa\sigma = (\kappa\ (\sigma\ (\rho\ \nu))\ \sigma)$$

$$\mathcal{E}[\![\texttt{(if}\ \pi\ \pi_1\ \pi_2\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\ \textbf{if}\ (\mathit{boolify}\ \varepsilon)$$
$$\textbf{then}\ \mathcal{E}[\![\pi_1]\!]$$
$$\textbf{else}\ \mathcal{E}[\![\pi_2]\!]$$
$$\textbf{endif}\ \ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(set!}\ \ \nu\ \pi\texttt{)}]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\kappa\ \varepsilon\ \sigma_1[(\rho\ \nu) \rightarrow \varepsilon])\ \sigma)$$

$$\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\kappa\ \mathit{in}\textbf{Value}(\lambda\ \varepsilon^*\delta_1\kappa_1\sigma_1.$$
$$\textbf{if}\ \#\varepsilon^* = \#\nu^*$$
$$\textbf{then}\ \ \mathit{allocate}\ \sigma_1\ \#\nu^*$$
$$\lambda\ \sigma_2\alpha^*.$$
$$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \delta_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])$$
$$\textbf{else}\ \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\ \sigma)$$

$$\mathcal{E}[\![\texttt{(}\pi\ \pi^*\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\ \varphi\sigma_1.$$
$$(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\ \varepsilon^*\sigma_2.$$
$$(\ \varphi\ |_{\textbf{Function}}$$
$$\varepsilon^*\ \delta\ \kappa\ \sigma_2)\ \ \sigma_1)\ \ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \delta\ \lambda\varepsilon^*\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\delta\kappa\sigma = (\kappa\ \langle\rangle\ \sigma)$$

$$\mathcal{E}[\![\texttt{(begin}\ \pi^+\texttt{)}]\!] = \mathcal{E}^+[\![\pi^+]\!]$$

$$\mathcal{E}^+[\![\pi]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\delta\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta\ \kappa\ \sigma_1)\ \sigma)$$

$$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$$
$$\mathit{in}\textbf{Value}(\lambda\ \varepsilon^*\delta\kappa\sigma.$$
$$\textbf{if}\ 1 = \#\varepsilon^*$$
$$\textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textbf{Function}}$$
$$\langle\mathit{in}\textbf{Value}(\lambda\ \varepsilon^*{}_1\delta_1\kappa_1\sigma_1.$$
$$\textbf{if}\ 1 = \#\varepsilon^*{}_1$$
$$\textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \sigma_1)$$
$$\textbf{else}\ \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\rangle\ \delta\ \kappa\ \sigma)$$
$$\textbf{else}\ \mathit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )$$

Table 5.3 Scheme and dynamic binding

The dynamic environment, identified by δ, follows a very different path from the lexical environment ρ because it is passed as an argument to functions that can exploit the current dynamic environment that way. It is also different from memory, indicated by σ, which is *single-threaded*: every step of a computation takes the current memory as input, consults it or modifies it, and passes it to the next step. Memory is thus unique since we never need the preceding version of it. The dynamic environment is not like memory because, for example, it is common to all the terms of a functional application that share it.

The dynamic environment is defined by the domain **DynEnv** like this:

$$\delta \quad \textbf{DynEnv} \quad = \textbf{Identifier} \rightarrow \textbf{Value}$$

There are two special forms for handling dynamic bindings: `dynamic-let` and `dynamic`. `dynamic-let` establishes a dynamic binding between a *variable* and a *value* while its *body* is being computed. `dynamic-let` knows how to handle only one unique variable, but that fact does not lessen the power of the language, nor even its ease of use. `dynamic` returns the value of a dynamic variable; it raises an error if the variable has not yet been defined. Since we have not defined any way of modifying such a binding, the dynamic environment directly associates the name of variables with their values without any intermediate addresses. You can see the semantics of these two special forms in Table 5.4. Here's their syntax:

`(dynamic-let (`*variable value*`)` *body* ...`)`
`(dynamic` *variable*`)`

$$(\delta_0\ \nu) = \mathit{wrong}\ \text{“}\texttt{No such dynamic variable}\text{”}$$

$$\mathcal{E}[\![\texttt{(dynamic}\ \nu\texttt{)}]\!]\rho\delta\kappa\sigma = (\kappa\ (\delta\ \nu)\ \sigma)$$

$$\mathcal{E}[\![\texttt{(dynamic-let (}\nu\ \pi\texttt{)}\ \pi^+\texttt{)}]\!]\rho\delta\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \delta\ \lambda\varepsilon\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \delta[\nu \rightarrow \varepsilon]\ \kappa\ \sigma_1)\ \sigma)$$

Table 5.4 Special forms of dynamic binding

Those two denotations are straightforward. They enlarge or search the dynamic environment and thus provide a new name space, one for dynamic variables—those that must not be confused with truly lexical variables. Once again, you can see from a short example using only a few Greek letters how powerful denotations are. They lend their aptitude to anyone who can decypher them.

The idea of a dynamic environment is very important. Not only does it serve dynamic variables; it also works for functions that handle errors, for escapes with dynamic extent, or for concepts that control computations, as in [HD90, QD93]. Here, for example, is a simplistic (and not very good) protocol for trapping errors: every time an anomaly occurs, the implementation constructs an object describing the anomaly and applies the value of the dynamic variable `*error*` to it. In that way, we protect ourselves from any possible error by means of the form `dynamic-let` binding `*error*` to an *ad hoc* function. As an example of this protocol, Scheme specifies that an attempt to open a non-existing file raises an error

but provides no function to know whether the file exists. We could then program a predicate indicating whether a file exists at the exact moment that the predicate is called, like this:

```
(define (probe-file filename)
  (dynamic-let (*error* (lambda (anomaly) #f))
    (call-with-input-file filename
      (lambda (port) #t) ) ) )
```

That's just an approximate definition because, in fact, we have to test whether the anomaly really has to do with the error "absent file" and not, for example, with the error "no more input ports available." To handle such errors correctly, an evaluation must cooperate by constructing the right objects to represent anomalies.

5.7 Global Environment

In this section, we'll study the global environment, denoting it in several different ways. Just as we did when we introduced dynamic bindings, we'll denote the essence of Scheme by adding a global environment. Like the local environment ρ, this global environment γ transforms identifiers (names) into addresses. The global environment will faithfully accompany memory: every step of a computation will return memory and a global environment, possibly one that has been modified. Table 5.5 defines an essential Scheme with an explicit global environment. However, we've removed denotations involved with reference to a variable and to assignment from this definition.

5.7.1 Global Environment in Scheme

On this basis, we'll be able to construct many possible definitions of the global environment. Scheme stipulates that *(i)* we can get the value of a variable only if it exists and is initialized; *(ii)* we can modify a variable only if it exists; *(iii)* redefining a variable is comparable to assignment.

To express those first two rules denotationally, we have to be able to test whether or not a variable appears in an environment. Thus instead of the codomain of environments being the address space, we'll enlarge this codomain with a point that differs from addresses. That point will denote the absence of a variable. In denotational terms, we'll do this:

$$\begin{aligned} \textbf{LocalEnvironment}: &\quad \textbf{Identifier} \to \textbf{Address} + \{\textit{no-such-binding}\} \\ \textbf{GlobalEnvironment}: &\quad \textbf{Identifier} \to \textbf{Address} + \{\textit{no-such-global-binding}\} \end{aligned}$$

As a consequence now, the denotations for referring to a variable and for assigning a variable are straightforward: first, we check whether the variable is local, then whether it is global. This check produces an address which we then use to read the value of the variable. Of course, the initial environments have to match this coding.

$(\rho_0\ \nu) = \textit{no-such-binding}$

$(\gamma_0\ \nu) = \textit{no-such-global-binding}$

$$\mathcal{E}[\![(\texttt{if}\ \pi\ \pi_1\ \pi_2)]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.\ \textbf{if}\ (\textit{boolify}\ \varepsilon)$$
$$\textbf{then}\ (\mathcal{E}[\![\pi_1]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)$$
$$\textbf{else}\ (\mathcal{E}[\![\pi_2]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)$$
$$\textbf{endif}\ \ \sigma)$$

$$\mathcal{E}[\![(\texttt{lambda}\ (\nu^*)\ \pi^+)]\!]\rho\gamma\kappa\sigma =$$
$$(\kappa\ \textit{in}\mathbf{Value}(\lambda\ \varepsilon^*\gamma_1\kappa_1\sigma_1.$$
$$\textbf{if}\ \#\varepsilon^* = \#\nu^*$$
$$\textbf{then}\ \ \textit{allocate}\ \sigma_1\ \#\nu^*$$
$$\lambda\ \sigma_2\alpha^*.$$
$$(\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma_1\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*])$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\ \gamma\ \sigma)$$

$$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varphi\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma_1\ \lambda\varepsilon^*\gamma_2\sigma_2.(\varphi\ |_{\textsc{Function}}\ \varepsilon^*\ \gamma_2\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\pi\ \pi^*]\!]\rho\gamma\kappa\sigma =$$
$$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^*[\![\pi^*]\!]\ \rho\ \gamma\ \lambda\varepsilon^*\gamma_2\sigma_2.(\kappa\ \langle\varepsilon\rangle\S\varepsilon^*\ \gamma_2\ \sigma_2)\ \sigma_1)\ \sigma)$$

$$\mathcal{E}^*[\![\ ]\!]\rho\gamma\kappa\sigma = (\kappa\ \langle\rangle\ \gamma\ \sigma)$$

$$\mathcal{E}[\![(\texttt{begin}\ \pi^+)]\!]\rho\gamma\kappa\sigma = (\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \kappa\ \sigma)$$

$$\mathcal{E}^+[\![\pi\ \pi^+]\!]\rho\gamma\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.(\mathcal{E}^+[\![\pi^+]\!]\ \rho\ \gamma_1\ \kappa\ \sigma_1)\ \sigma)$$

$$(\sigma_0(\rho_0[\![\texttt{call/cc}]\!])) =$$
$$\textit{in}\mathbf{Value}(\lambda\ \varepsilon^*\gamma\kappa\sigma.$$
$$\textbf{if}\ 1 = \#\varepsilon^*$$
$$\textbf{then}\ (\ \varepsilon^*\downarrow_1|_{\textsc{Function}}$$
$$\langle\textit{in}\mathbf{Value}(\lambda\ \varepsilon^*{}_1\gamma_1\kappa_1\sigma_1.$$
$$\textbf{if}\ 1 = \#\varepsilon^*{}_1$$
$$\textbf{then}\ (\kappa\ \varepsilon^*{}_1\downarrow_1\ \gamma_1\ \sigma_1)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )\rangle\ \gamma\ \kappa\ \sigma)$$
$$\textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”}$$
$$\textbf{endif}\ \ )$$

Table 5.5 Scheme and a global environment

$\mathcal{E}[\![\nu]\!] =$
$\lambda\rho\gamma\kappa\sigma.$ **let** $\alpha = (\rho\ \nu)$
in **if** $\alpha =$ *no-such-binding*
then **let** $\alpha_1 = (\gamma\ \nu)$
in **if** $\alpha_1 =$ *no-such-global-binding*
then *wrong* "`No such variable`"
else $(\kappa\ (\sigma\ \alpha_1)\ \gamma\ \sigma)$
endif
else $(\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma)$
endif

$\mathcal{E}[\![$`(set!` $\nu\ \pi$`)`$]\!]\rho\gamma\kappa\sigma =$
$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.$ **let** $\alpha = (\rho\ \nu)$
in **if** $\alpha =$ *no-such-binding*
then **let** $\alpha_1 = (\gamma_1\ \nu)$
in **if** $\alpha_1 =$ *no-such-global-binding*
then *wrong* "`No such variable`"
else $(\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \rightarrow \varepsilon])$
endif
else $(\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon])$
endif $\sigma)$

We assume that the definition of global variables is carried out by the special operator `define`; also we assume that the special form `(define` $\nu\ \pi$`)` defines (or redefines) the *global* variable ν. For the moment, we'll ignore internal definitions (introduced by the internal forms `(define ... )`); they are purely syntactic; for that reason, we'll appoint `define-global` for external definitions. Here, then, is the denotation of a definition, whether the effect of introducing a new variable or the effect of modifying an existing one.

$\mathcal{E}[\![$`(define-global` $\nu\ \pi$`)`$]\!]\rho\gamma\kappa\sigma =$
$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.$ **let** $\alpha = (\gamma\ \nu)$
in **if** $\alpha =$ *no-such-global-binding*
then *allocate* $\sigma_1\ 1\ \ \lambda\sigma_2\alpha^*.(\kappa\ \varepsilon\ \gamma_1[\nu \rightarrow \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \rightarrow \varepsilon])$
else $(\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon])$
endif $\sigma)$

These definitions characterize the behavior specified in Scheme except for the definition of the initial global environment γ_0, which we've left a little vague; it could contain only standard variables, or it could add a few additional ones, or it might even contain every imaginable variable that we might ever need. In that latter case, it would not know about non-existing variables, but we might encounter uninitialized variables.

5.7.2 Automatically Extendable Environment

Certain Lisp systems let the first assignment of a free variable be equivalent to its definition. It's easy to modify the semantics of assignment to incorporate the effect of a definition if the variable does not yet exist.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(set!}\ \ \nu\ \pi\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\ \varepsilon\gamma_1\sigma_1. \\
\qquad\qquad \textbf{let}\ \alpha = (\rho\ \nu) \\
\qquad\qquad \textbf{in}\ \ \textbf{if}\ \alpha = \textit{no-such-binding} \\
\qquad\qquad\qquad \textbf{then}\ \ \textbf{let}\ \alpha_1 = (\gamma_1\ \nu) \\
\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \textbf{if}\ \alpha_1 = \textit{no-such-global-binding} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\kappa\ \varepsilon\ \gamma_1[\nu \rightarrow \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \rightarrow \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha_1 \rightarrow \varepsilon]) \\
\qquad\qquad\qquad\qquad\qquad \textbf{endif} \\
\qquad\qquad\qquad \textbf{else}\ (\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon]) \\
\qquad\qquad\qquad \textbf{endif}\quad \sigma)
\end{array}
$$

Such a variation is useful because the form **define** is no longer necessary and can be conflated with **set!**. The disadvantage is that a misspelling of the name of a variable can be hard to find since it does not automatically lead to an error at its first use.

5.7.3 Hyperstatic Environment

Inside a hyperstatic global environment, functions enclose not only the local definition environment but also the current global environment. We express that idea straightforwardly by a simple change in the denotation of an abstraction as it appeared in Table 5.5.

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(lambda (}\nu^*\texttt{)}\pi^+\texttt{)}]\!]\rho\gamma\kappa\sigma = \\
(\kappa\ \textit{in}\textbf{Value}(\lambda\ \varepsilon^*\gamma_1\kappa_1\sigma_1. \\
\qquad\qquad \textbf{if}\ \#\varepsilon^* = \#\nu^* \\
\qquad\qquad \textbf{then}\ \ \textit{allocate}\ \sigma_1\ \#\nu^* \\
\qquad\qquad\qquad\qquad \lambda\ \sigma_2\alpha^*. \\
\qquad\qquad\qquad\qquad\quad (\mathcal{E}^+[\![\pi^+]\!]\ \rho[\nu^* \xrightarrow{*} \alpha^*]\ \gamma\ \kappa_1\ \sigma_2[\alpha^* \xrightarrow{*} \varepsilon^*]) \\
\qquad\qquad \textbf{else}\ \textit{wrong}\ \text{“}\texttt{Incorrect arity}\text{”} \\
\qquad\qquad \textbf{endif}\quad)\ \gamma\ \sigma)
\end{array}
$$

That semantics is compatible with assignment in Scheme, where assignment can modify only existing variables. However, defining new variables on the fly by assignment leads to confusion. Just consider this:

```
(define (weird v)
  (set! a-new-variable v) )
```

The assignment inside the function **weird** extends the global environment that it closes, and it does so at every invocation since that extended global environment is not stored by **weird**.

As we've already mentioned, the problem with hyperstatic global environments is that non-local mutually recursive functions cannot be defined there straightforwardly. We could introduce a new form for codefinitions, authorizing the cojoint definition of a multitude of bindings, but we would rather introduce the possibility of referencing the global environment by means of a new special form **(global** ν**)**.

That will make it possible to reference the global variable ν in *any* context, even if a local variable ν exists. As a consequence, we can write the codefinition of the functions `odd?` and `even?` like this:

```
(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?) )
```

$\mathcal{E}[\![$`(global` ν`)`$]\!]\rho\gamma\kappa\sigma =$
let $\alpha = (\gamma\ \nu)$
in **if** $\alpha =$ *no-such-global-binding*
then *wrong* "`No such variable`"
else $(\kappa\ (\sigma\ \alpha)\ \gamma\ \sigma)$
endif

$\mathcal{E}[\![$`(define-global` $\nu\ \pi$`)`$]\!]\rho\gamma\kappa\sigma =$
$(\mathcal{E}[\![\pi]\!]\ \rho\ \gamma\ \lambda\varepsilon\gamma_1\sigma_1.$ **let** $\alpha = (\gamma\ \nu)$
in **if** $\alpha =$ *no-such-global-binding*
then *allocate* $\sigma_1\ 1\ \ \lambda\sigma_2\alpha^*.(\kappa\ \varepsilon\ \gamma_1[\nu \rightarrow \alpha^*\downarrow_1]\ \sigma_2[\alpha^*\downarrow_1 \rightarrow \varepsilon])$
else $(\kappa\ \varepsilon\ \gamma_1\ \sigma_1[\alpha \rightarrow \varepsilon])$
endif $\sigma)$

There we see again that denotational semantics enables us to specify many diverse environments of a language very elegantly. Of course, we can combine the various environments explicated here so that we could offer multiple name spaces in all their specificity, like COMMON LISP does.

5.8 Beneath This Chapter

The main purpose of this chapter was to demystify denotational semantics. We've gone about this in an informal way (some might say in a sacrilegious way) because that seemed to us the likeliest means of stimulating wider use of denotational semantics. As we have progressively enriched the linguistic characteristics specified by our various interpreters and no less progressively reduced the definition language, we've been able gradually to move toward the denotations in this chapter, or perhaps we should say toward a denotational interpreter. Even though Scheme and λ-calculus hardly seem alike, they are not so very dissimilar. In fact, it is generally possible to get an executable denotation to correct errors that infiltrate these semantic equations. The fact that it's executable is highly important: that's what makes it possible for the defined language to behave like the designer wants it to. The designer can exercise the language, experiment with it, test it to insure its conformity with the equations of his or her dreams. Since the denotation is immediately executable as it appears in the equations, we can skip the implementation phase where we would ordinarily have to prove rigorously that no distortions have crept into the implementation.

Having to write a denotational interpreter is thus a gratifying and reassuring activity. It doesn't cost us any advantages of λ-calculus, but it adds a supplementary constraint: a denotational interpreter has to be executable for a call by value

(that "bad" evaluation method that Scheme uses). This constraint can often be satisfied; to witness: all the denotations in this chapter correspond to code written in Scheme and in fact they have passed through a little converter (LiSP2TEX) to print them in Greek [Que93d]. Just imagine what it would be like to write the denotation of the application preserving the quality that evaluation order makes no difference if we were not using an applicative language in which we can test such functions!

Here's an example of the denotation of a simple abstraction (without variable arity) so you can compare its "Greek" portrait on page 160.

```
(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 ) )
```

Here we've used the outmoded form **define** that allowed a first argument in the call position. The form **(define (**ν . *variables*) π^***)** is recursively equivalent to **(define** ν **(lambda** *variables* π^***))** where ν can still be a form of calling.

We've articulated denotations case by case, automatically carried out by an appropriate function which produces the syntactic analysis. By now you're familiar with its structure:

```
(define (meaning e)
  (if (atom? e)
      (if (symbol? e) (meaning-reference e)
                      (meaning-quotation e) )
      (case (car e)
        ((quote)  (meaning-quotation (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-assignment (cadr e) (caddr e)))
        (else     (meaning-application (car e) (cdr e))) ) ) )
```

5.9 λ-calculus and Scheme

Since we are concerned about whether denotations can be executed, we have a problem about the difference between Scheme and λ-calculus. (Of course, we're considering only that subset of "pure" Scheme with no assignment, no side effects.) The difference rests mainly in the evaluation strategy. There is no fixed strategy for λ-calculus, so we have to be careful not to take a bad one. In contrast, there is an evaluation strategy for Scheme, and it's different. Scheme uses call by value.

That is, the arguments of an application are evaluated before they are submitted to the function. Moreover, Scheme does not evaluate bodies of **lambda** forms.

We can simulate call by name in Scheme by using thunks. A thunk is a function without variables that we also call a "promise" according to some terminologies. In Scheme, there is the syntax **delay** to construct a promise, that is, an object representing a certain computation to start (or **force**) once the time for it arrives. We define **delay** and **force** like this:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression) (lambda () expression)) ) )
(define (force promise) (promise))
```

The form **delay** closes an expression in its lexical context in a thunk that we invoke on demand by **force**. We simulate a call by value with this mechanism if we transform the program like this: every application $(f\ a\ \ldots\ z)$ is rewritten as $(f$ **(delay** $a)\ \ldots$ **(delay** $z))$, and every variable in the body of a function is unfrozen by means of **force**. Here's an example. This calculation used to be problematic in Scheme:

```
(((lambda (x) (lambda (y) y))                    ;((λx.λy.y (ωω)) z)
  ((lambda (x) (x x)) (lambda (x) (x x))) )
 z )
```

We rewrite that computation in the following way to suspend the computation indefinitely without changing it and to return the value of the free variable **z**, like this:

```
(((lambda (x) (lambda (y) (force y)))
  (delay ((lambda (x) ((force x) (delay (force x))))
          (lambda (x) ((force x) (delay (force x))))) ))
 (delay z) )
```

Although it's correct according to [DH92], this rewrite introduces some inefficiency. Putting aside the double promise in **(delay (force x))** (which is, in fact, equivalent to **x** which is already a promise) every time we need the value of a variable, we are obliged to force the computation even though it would suffice to do it only once and store the result. This technique is known as call by need. It corresponds to modifying **delay** to store the value that it leads to. Fortunately, Scheme is an excellent language for implementing languages, so we can program this technique by calling an assignment to the rescue, like this:

```
(define-syntax memo-delay
  (syntax-rules ()
    ((memo-delay expression)
     (let ((already-computed? #f)
           (value 'wait) )
       (lambda ()
         (if (not already-computed?)
             (begin
               (set! value expression)
               (set! already-computed? #t) ) )
         value ) ) ) ) )
```

Of course, we could refrain from carrying out these transformations by hand and design macros to do them, as in [DFH86]. If we really wanted to be efficient, we could go on with a *strictness* analysis to determine the cases where variables are always forced. That analysis would let us avoid constructing promises for those variables, as in [BHY87]. Finally, we could adopt clever code for the promises to eliminate the cost at execution time of the test `(if (not already-computed?) ...)`. In Scheme, promises let us behave as if we were working in a lazy language where transformations announced earlier will be carried out automatically. However, this programming style poses problems during debugging because the computation is distributed. Moreover, this style does not go well with assignments and continuations. Just compare these two programs from [KW90, Mor92]. They differ only by one sole `delay` but they compute very different results.

```
(pair? (call/cc (lambda (k) (list (k 33)))))
(pair? (call/cc (lambda (k) (list (delay (k 33))))))
```

5.9.1 Passing Continuations

Even a quick look at denotations makes us realize that they are written in the programming style known as *programming by continuations* or CPS for *Continuation Passing Style*. Realizing that all of Scheme, including `call/cc` can be denoted by λ-calculus, which can itself be seen as a certain subset of Scheme excluding `call/cc`, we might validly ask about the possibility of transforming programs automatically from Scheme to Scheme, just to get rid of `call/cc`.

The chief interest of this style is thus to make continuations appear explicitly. Realizing that these same continuations can be represented by `lambda` forms, we ask, *"Can we get along without the primitive* `call/cc`*?"* The answer is yes, and we can thus develop a transformation that converts a program with `call/cc` into another equivalent program without it. Some compilers even use this latter form as a quasi-intermediate language, as in [App92a], because it contains only the simplest syntactic constructions. Others don't like it because this form in CPS is no longer readable, and it fixes the order of evaluation too soon. Nevertheless, it is equivalent, as [SF92] shows. The version of this transformation that we'll show soon is strongly inspired by [DF90]. We'll use yet another version in Section 10.11.2. **[see p. 404]**

Let's look at the technique. We'll assume that every function will be called with a supplementary argument,[5] the continuation. Thus $_k$`(foo bar ... hux)` will be transformed into `(foo` *k* `bar ... hux)`. Consequently, we must have a representation of the continuation, we must also know how to handle special forms, so we'll go directly to an analysis of the text to translate, just as all the preceding interpreters have done. Like its denotational equivalent, the continuation waits for a value and returns the final result of the computation. The continuation thus closes the rest of the computations to carry out.

```
(define (cps e)
  (if (atom? e)
      (lambda (k) (k ',e))
      (case (car e)
```

5. We'll put it first to simplify the management of functions with variable arity.

```
((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) (caddr e)))
(else     (cps-application e)) ) ) )
```

By passing a continuation, the transformation takes a program and returns a closure that converts one program into another. In other words, the type of the **cps** function is this:

```
Program → (( Program → Program ) → Program )
```

Quoting is now easy to handle, too:

```
(define (cps-quote data)
  (lambda (k)
    (k '(quote ,data)) ) )
```

Assignment becomes this:

```
(define (cps-set! variable form)
  (lambda (k)
    ((cps form)
     (lambda (a)
       (k '(set! ,variable ,a)) ) ) ) )
```

It converts the form for which the value will serve as the assignment to the variable and inserts it in its place in the assignment which must be generated.

Conditional is handled similarly:

```
(define (cps-if bool form1 form2)
  (lambda (k)
    ((cps bool)
     (lambda (b)
       `(if ,b ,((cps form1) k)
               ,((cps form2) k) ) ) ) ) )
```

Sequence is comparable:

```
(define (cps-begin e)
  (if (pair? e)
      (if (pair? (cdr e))
          (let ((void (gensym "void")))
               (lambda (k)
                 ((cps-begin (cdr e))
                  (lambda (b)
                    ((cps (car e))
                     (lambda (a)
                       (k '((lambda (,void) ,b) ,a)) ) ) ) ) ) )
          (cps (car e)) )
      (cps '()) ) )
```

Notice that **begin** forms have been suppressed in favor of closures.

The complicated part is how to handle the form **lambda**. It must be converted into a new function that takes a supplementary variable, and the functional application must provide the continuation as a supplementary argument to the invoked function. A slight improvement has been introduced here when the called function

is trivial; that is, when it carries out a simple computation, a short one that always terminates. These functions have been organized into lists[6] of `primitives`.

```
(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 body)
              (lambda (a) `(,c ,a)) ) ) )) ) )
```

That transformation is complete now, and we can use it to experiment with the factorial, like this:

```
(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) ) ) ) )
```

Here we automatically get what we had to write by hand earlier. Notice that there are no complicated calculations after the transformation, only trivial applications like comparisons or simple arithmetic operations or applications with continuations. The rest is made up of only variables or closures.

In this world, a form $_k$`(call/cc f)` becomes `(call/cc` k `f)`. The function `call/cc` is no more than `(lambda (k f) (f k k))` and can be simplified directly so it does not show up anymore. Nevertheless, we must bind the global variable

6. Correct treatment of these calls to predefined primitive functions will be covered in Section 6.1.8.

`call/cc` to `(lambda (k f) (f k k))` in such a way that we can write, for example, `(procedure? (apply call/cc (list call/cc)))`.

One virtue of a program written in CPS is that since we have put everything totally in sequence by explicitly indicating when terms should be evaluated and to whom their values should be returned, it is completely insensitive to the evaluation strategy. We could evaluate with call by value or call by name; we would get the same results. Thus by combining promises and CPS we can patch up the differences between Scheme and λ-calculus as in [DH92].

5.9.2 Dynamic Environment

The preceding section showed how the denotation of `call/cc` enables us to invent a program transformation that eliminates this very `call/cc`. With the denotation of the dynamic environment, we can imagine applying the same technique to get rid of the special forms `dynamic` and `dynamic-let`. To do so, we simply have to introduce the dynamic environment explicitly everywhere it's needed.

Let's assume we have an identifier that shows up nowhere else. Let's call it δ. The dynamic environment will be the value of δ everywhere, and it will be represented by a function transforming symbols that name dynamic variables into values. Let's also assume that the function `update` extends an environment functionally [**see p. 129**], that it's available everywhere, and that it cannot be hidden. The transformation known as $\mathcal{D}$ and the utility $\mathcal{D}^*$ appear in Table 5.6.

$\mathcal{D}^*[\![]\!]$	$\rightarrow$
$\mathcal{D}^*[\![\pi\ \pi^*]\!]$	$\rightarrow$ $\mathcal{D}[\![\pi]\!]\ \mathcal{D}^*[\![\pi^*]\!]$
$\mathcal{D}[\![$`(if` $\pi_0\ \pi_1\ \pi_2$`)`$]\!]$	$\rightarrow$ `(if` $\mathcal{D}[\![\pi_0]\!]\ \mathcal{D}[\![\pi_1]\!]\ \mathcal{D}[\![\pi_2]\!]$`)`
$\mathcal{D}[\![$`(begin` π^*`)`$]\!]$	$\rightarrow$ `(begin` $\mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(`$\pi\ \pi^*$`)`$]\!]$	$\rightarrow$ `(`$\mathcal{D}[\![\pi]\!]\ \delta\ \mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(lambda (`ν^*`)` π^*`)`$]\!]$	$\rightarrow$ `(lambda (`$\delta\ \nu^*$`)` $\mathcal{D}^*[\![\pi^*]\!]$`)`
$\mathcal{D}[\![$`(dynamic` ν`)`$]\!]$	$\rightarrow$ `(`δ `(quote` ν`))`
$\mathcal{D}[\![$`(dynamic-let (`$\nu\ \pi$`)` π^+`)`$]\!]$	$\rightarrow$ `(let ((`δ `(update` δ `(quote` ν`)` π`)))` $\mathcal{D}^*[\![\pi^+]\!]$`)`

Table 5.6 Transformation suppressing the dynamic environment

That transformation simply simulates dynamic bindings if we have no other means. The ultimate detail is to specify the initial dynamic environment. The program π to transform it will thus be:

`(let ((`δ `(lambda (n) (error "No such dynamic variable" n)))) `$\mathcal{D}[\![\pi]\!]$`)`

We can sum up our efforts this way: many denotations lead to program transformations that eliminate the forms that they define.

5.10 Conclusions

This chapter culminates a series of interpreters that more and more precisely define a language in the same class as Scheme; they do so with more and more restricted means. Denotational semantics, at least as we have explored it in this chapter, is a remarkably concise way of defining the *kernel* of a functional language. It often

seems maladapted and unduly complicated to define an entire language in all its least details. The way we have presented Scheme here rules out the comparison of functions, the denotation of constants, and all sorts of characteristics that strain a definition with minutiae that are useful only rarely. Denotational semantics appears at its best when it outlines the general shape of a language, but it is quite boring for describing every single detail.

The range of things that we can define denotationally is quite vast. We can introduce parallelism with the technique of step calculations as in [Que90c]. We can define the effects of distributed data as in [Que92b]. There are also limitations, as indicated in [McD93]. For example, there is no easy way to define type inference denotationally, and that is one of the reasons for natural semantics [Kah87].

5.11 Exercises

Exercise 5.1 : Here is another way of writing the denotation of a functional application. Show that it is still equivalent to the one in Section 5.2.6.

$\mathcal{E}[\![(\pi\ \pi^*)]\!]\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varphi\sigma_1.(\overline{\mathcal{E}}[\![\ \pi^*]\!]\ \langle\rangle\ \rho\ \lambda\varepsilon^*\sigma_2.(\varphi\ |_{\text{Function}}\ \varepsilon^*\ \kappa\ \sigma_2)\ \sigma_1)\ \sigma)$

$\overline{\mathcal{E}}[\![\ ]\!]\varepsilon^*\rho\kappa\sigma = (\kappa\ (\mathit{reverse}\ \varepsilon^*)\ \sigma)$

$\overline{\mathcal{E}}[\![\pi\ \pi^*]\!]\varepsilon^*\rho\kappa\sigma = (\mathcal{E}[\![\pi]\!]\ \rho\ \lambda\varepsilon\sigma_1.(\overline{\mathcal{E}}[\![\ \pi^*]\!]\ \langle\varepsilon\rangle\S\varepsilon^*\ \rho\ \kappa\ \sigma_1)\ \sigma)$

Exercise 5.2 : It is not very efficient to define recursive functions within λ-calculus the way we did in Section 5.3. Instead, we could enrich the language with the special form `label`, like in Lisp 1.5. In terms of Scheme, the form `(label` ν `(lambda ...))` is equivalent to `(letrec ((`ν `(lambda ...))) `ν`)`. Define the semantics of this `label` operator.

Exercise 5.3 : Modify the denotation of the special form `dynamic` so that if the dynamic variable is not found, its value in the global environment will be returned.

Exercise 5.4 : Write a macro to simulate a non-prescribed evaluation order in an implementation of Scheme that evaluates from left to right. You may assume that there is a unary function `random-permutation` that takes an integer n and returns a random permutation of $1, \ldots, n$.

Recommended Reading

The readings that you mustn't miss are [Sto77] and [Sch86]. Both are mines of information about denotational semantics and present examples of denotations of the language.

Serious fans of λ-calculus should also read [Bar84].

6
Fast Interpretation

In the preceding chapter, there was a denotational interpreter that worked with extreme precision but remarkably slowly. This chapter analyzes the reasons for that slowness and offers a few new interpreters to correct that fault by pretreating programs. In short, we'll see a rudimentary compiler in this chapter. We'll successively analyze: the representation of lexical environments, the protocol for calling functions, and the reification of continuations. The pretreatment will identify and then eliminate computations that it judges static; it will produce a result that includes only those operations that it thinks necessary for execution. Specialized combinators are introduced for that purpose. They play the role of an intermediate language like a set of instructions for a hypothetical virtual machine.

The denotational interpreter of the preceding chapter culminated a series of interpreters leading to inexorably increasing precision. Now we'll have to correct that unbearable slowness. Still adhering to our technique of incremental modifications, particularly because the preceding denotational interpreter is the linguistic standard we have to conform to, we will present three successive interpreters, gradually relaxing some of the preliminary descriptive concerns for the benefit of the habits and customs of implementers.

6.1 A Fast Interpreter

To produce an efficient interpreter now, we'll assume that the implementation language contains a minimal number of concepts, notably, memory. We'll get rid of the one we added in Chapter 4 [see p. 111] since we added it just to explain the idea of memory. The only part we will keep has to do with binding variables. That activity delegates the management of dotted pairs to `cons`, the management of vectors to `vector`, the management of closures to `lambda`, and so on down the line. Memory will no longer be a huge mixture of function arguments, data structures, and even labels on closures. In fact, memory will now contain only bindings or activation records.

6.1.1 Migration of Denotations

The main source of inefficiency in our denotational interpreter is that programs are ceaselessly denoted, denoted again, and again and again. The repair is simple: we need to move the computations into positions where they will be calculated as soon as possible, as in [Deu89]. Moving this way is known as *migrating* code and as *λ-hoisting* in [Tak88] or as *λ-drifting* in [Roz92]. We must be careful in applying it to Scheme because of side effects that might occur at an inappropriate time or even the wrong number of times. A related problem is that migrating code can also change the termination of an entire program if the computation being migrated is one that loops indefinitely. Indeed, the expression `(lambda (x) (ω ω))` where `(ω ω)` is a non-terminating computation, is not equivalent to `(let ((tmp (ω ω))) (lambda (x) tmp))`. However, these reasons don't hold for denotations because denotations are exempt from side effects. Moreover, the denotations that migrate always terminate because denotations are compositional, and the programs being treated are always finite trees.

As an example of migration, here's a new version of abstraction. Here you can see that the denotation of the body of the abstraction `(meaning*-sequence e+)` is calculated only once, as is `(length n*)`, the number of variables in the abstraction.

```
(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 Activation Record

If we decrease memory consumption, then in large measure we will also improve interpretation speed, too. One major reason for memory consumption is the function calling protocol as expressed in the denotational interpreter. Invoking a function there means providing it values that will become its arguments, that is, the values of its variables. The denotational interpreter provided these values as a list; these same values were then concatenated to the environment, which was also represented as a list. Since it is easy to write this way, the fact that we were specifying and prototyping could justify our use of so many lists, but any serious pursuit of speed rules out this way of working.

While some of these allocations are superfluous, others cannot be avoided. It's a natural reflex of every implementer to exploit these latter to produce the effects of the former as well. Since providing values to a function is the same act as invoking

it, we can't really eliminate it, and besides, those values have to appear somewhere: in registers, in a stack, or even in an *activation record*. An activation record will be represented by an object that contains the values provided to a function, so temporarily, we can do this:

```
(define-class activation-frame Object                    TEMPORARY
  ( (* argument) ) )
```

That definition exploits a new characteristic of our object system that we haven't mentioned before: the star '*' specifies that a field is *indexed*; that is, it is not reduced to a single value but instead contains an ordered sequence of a size determined at creation time. (See Section 11.2 for more details about that idea.)

That representation is more advantageous than representation as a list once the number of arguments is greater than two. Moreover, it supports direct access to arguments, rather than sequential, linear access as in lists. An activation record must allow a lexical environment to be extended. Of course, we could even do something so that an environment would be a list of activation records, as in Figure 6.1

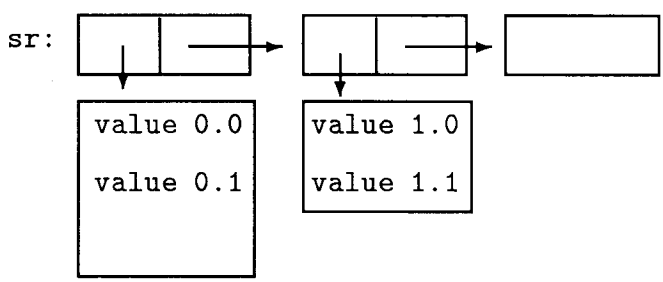


Figure 6.1 Environment and lists of activation records

Even more happily, we could reserve a supplementary field in activation records to link them together. Since the time needed to allocate activation records hardly depends on their size (as long as we overlook the memory management problems of initializing fields), we exchange two allocations for one enlarged only by a pointer. For those reasons, we'll adopt the following definition for activation records where we insert the field for linking in first position so we can follow it or modify it by an offset without disturbing any of the arguments that follow it, as in Figure 6.2.

```
(define-class environment Object
  ( next ) )
(define-class activation-frame environment
  ( (* argument) ) )
```

The values present in lexical environments are thus linked together in a chain of activation records. The function **sr-extend*** does this linking, like this:

```
(define (sr-extend* sr v*)
  (set-environment-next! v* sr)
  v* )
```

Activation records are physically modified. For that reason, they cannot be re-used since the functional invocation must allocate fresh bindings.

If we think again about the presentation of lexical environments in the denotational interpreter, we recall that they were composed of two distinct parts, ρ and σ. The environment ρ associates the name of a variable with an address so we can determine its value in the memory σ. But here we've saved little from memory except management of activation records. Any value can be retrieved from the chain of activation records by an "address" made up of two numbers: the first indicates in which activation record to look; the second then tells the index of the argument to look for. Figure 6.2 and the following two functions for reading and writing illustrate that idea.

```
(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) ) )
```

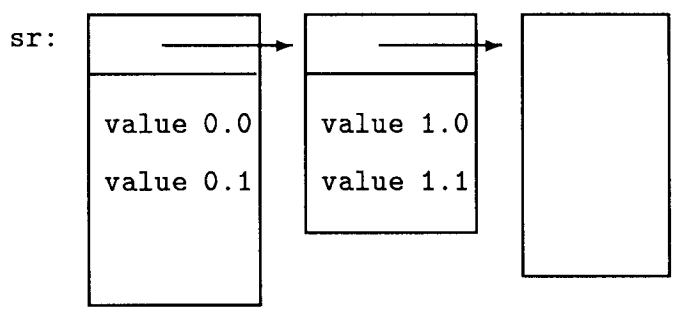


Figure 6.2 Environment and linked activation records

Thus the environment associating an identifier with an address enables us to retrieve the associated value. Memory (which we are trying to get out of our new interpreter) now exists for the sole purpose of representing activation records. Consequently, we'll still use `r` as the name for lexical environments, but we'll adopt `sr` for the *store* associated with the environment `r`, that is, the representation at execution of values in this environment. Although this latter should be represented by a linked list of activation records, we'll adopt a "ribcage" representation for the environment. That is, we'll use the list of lists of variables from abstractions. [see **Ex. 1.3**] We'll extend the environment by `r-extend*` (not to be confused with `sr-extend*`), and we'll use a semi-predicate `local-variable?` to search for the *lexical indices* of a value.

```
(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)) ) ) ) )
```

The purpose of doing things this way is to get a strong block structure from activation records. The strong block structure means that the address associated with an identifier is no longer an absolute address, but rather a pair of numbers interpreted relative to the linked activation records. As a consequence, computations with addresses become static calculations that can migrate during pretreatment of expressions to evaluate.

Cutting the lexical environment this way into static and dynamic parts does not depend on the representation that we have just chosen. It's actually a deeper property that only now becomes apparent. With the representation we chose for the denotational interpreter, it was already possible to predict the "position" (where position is counted in number of comparisons by **eq?**) of any variable inside the environment since the environment itself was merely a chain of the closures of bodies `(lambda (u) (if (eq? u x) y (f u)))`.

6.1.3 The Interpreter: the Beginning

We know enough now to show the skeleton of the interpreter and a few of its special forms. As you might expect, we begin with the function **meaning**. It will now have the following signature:

$$\mathbf{meaning}: \underbrace{\mathbf{Program} \times \mathbf{Environment}}_{static} \rightarrow \underbrace{(\mathbf{Activation\text{-}Record} \times \mathbf{Continuation} \rightarrow \mathbf{Value})}_{dynamic}$$

This signature clearly shows how the environment is cut into its static and dynamic components, **Environment** and **Activation-Record**. When a program is pretreated, the result is a binary function waiting for a list of linked activation records (that is, memory) and a continuation to calculate a value. This is a nonstandard way of representing programs, far removed from the original S-expression, but fundamentally the same structurally.

To simplify syntactic analysis, we'll assume as usual that the expressions submitted are syntactically legitimate programs. Here are the rules we'll use to name variables when we define the functions for syntactic analysis:

e ...	expression, form
r ...	environment
sr, ... , v* ...	activation record
v ...	value (integer, pair, closure, etc.)
k ...	continuation
f ...	function
n ...	identifier

Here, then, are the functions for analyzing syntax:

```
(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)) ) ) )
```

Once again, quoting becomes trivial, like this:

```
(define (meaning-quotation v r)
  (lambda (sr k)
    (k v) ) )
```

Here's the conditional, a good example of code migration. We pretreat the two branches of the conditional, regardless of which of them will be the choice that might be made.

```
(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)) ) ) ) )
```

We decompose a sequence into two conventional subcases, like this:

```
(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) )) ) ) )
```

Things get a little stickier with an application. For an application, we have to define the invocation protocol more precisely.

Application

To pretreat an application, we must make several things explicit: how it's created, how it's filled in, the way it's passed—in short, how activation records handle it. While the pretreatment of the function term is conventional enough, how the arguments are handled is less so. For that purpose, we'll use the function `meaning*`. For simplicity, functions will be represented by their closures.

```
(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) ) )) ) ) )
```

Since we've been looking for computations that we can do statically,[1] we've seen that the size of an activation record to allocate is easily predicted since it can be deduced directly from the number of terms in the application. In contrast, it's much harder to know *when* to allocate the record. There are two potential moments: [see Ex. 6.4]

1. We could allocate the record before evaluating its arguments. In that case, each argument calculated there is immediately put into place.
2. We could allocate the record after evaluating its arguments. In that case, however, we consume twice as much memory since we have to store the values in the continuation—the same values that will all be organized into the newly allocated record.

The first of those two strategies seems more efficient since it consumes less memory. Unfortunately, the presence of `call/cc` in Scheme totally ruins that possibility. It's feasible only for Lisp. The reason: in Scheme it's possible to call a continuation more than once. [see p. 82] If the record is allocated first, before the arguments are computed, then, if one of those computations captures its continuation, it will also capture the record that appears in the continuation. The record will thus be shared by all the invocations of the function term—a sharing that is contrary to the abstraction which must allocate new addresses for its local variables at every invocation. The following program should return `(2 1)`. Sharing activation records would incorrectly force a return of `(2 2)`.[2] In effect, the

1. By the way, that's a major activity among language designers; they actually favor characteristics that are static.
2. Manuel Serrano discovered that a previous version of this example was depending subtly on the order of evaluation. The form `cons` should be evaluated from left to right. The form `(let ((g ... ))... )` does that.

form `call/cc` captures the application `((lambda (a) ... ) ... )` and notably the activation record if it has been pre-allocated. Since that continuation is used twice, the two thunks created by `(lambda () a)` share the closed variable `a` by conferring on it the last of the values that `k` received.

```
(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 (lambda () 1))
  (if (null? (cdr f)) (k 2))
  ;; f ≈(list (lambda () 2) (lambda () 1))
  (list ((car f)) ((cadr f))) )
```

But in fact, all is not lost in Scheme. It suffices in the implementation of `call/cc` to duplicate the activation records captured when the continuations were invoked. We can't program that here because continuations are represented by closures, from which we cannot extract the enclosed activation records. In that example, you can clearly see the impact of `call/cc`.

The function `meaning*` will thus take a supplementary argument corresponding to the size of the activation record that must be allocated afer the evaluation of all arguments. For reasons that will be clear when we discuss the implementation of functions with variable arity in Section 6.1.6, [see p. 196], activation records will contain one more field than necessary, but we will not initialize this excess field, so it won't penalize these functions.

```
(define (meaning* e* r size)
  (if (pair? e*)
      (meaning-some-arguments (car e*) (cdr e*) r size)
      (meaning-no-argument r size) ) )
(define (meaning-no-argument r size)
  (let ((size+1 (+ size 1)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

Notice that `size+1` is precalculated since it would be too bad to leave that computation until execution! Also notice the "non-migration" of the allocation form that has to allocate a new activation record at every invocation.

Each term of the application is put into the right place, just after allocation of the activation record. The right place is easily calculated in terms of the variables `size` and `e*`.

```
(define (meaning-some-arguments e e* r size)
  (let ((m (meaning e r))
        (m* (meaning* e* r size))
        (rank (- size (+ (length e*) 1))) )
    (lambda (sr k)
      (m sr (lambda (v)
              (m* sr (lambda (v*)
                       (set-activation-frame-argument! v* rank v)
                       (k v*) )) )) ) ) )
```

We can finally define abstractions since they appear so clear now. Verification of the arity is carried out by inspection of the size of the activation record. There again, we've precalculated everything we can so we leave as little as possible until execution.

```
(define (meaning-fix-abstraction n* e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
         (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 Classifying Variables

Those preceding definitions handle only the case of local variables, that is, only variables in **lambda** forms. There are, of course, global variables, and among them, predefined variables and/or immutable ones, like **cons** or **car**. In our current state, the only way of accessing them would be to follow the links between activation records, but that technique makes access to global variables particularly slow since they are located in the ultimate activation record. For that reason, we'll treat these statically classified variables differently.

We'll assume that the global variable **g.current** contains the list of mutable global variables, while **g.init** contains the list of predefined, immutable ones, such as **cons**, **car**, etc. The function **compute-kind** classifies variables and returns a descriptor characterizing variables.

```
(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)) ) )
```

We test **g.current** before **g.init** so that we can mask predefined variables, if need be. Considering primitives as values of immutable global variables is a safe practice. However, certain implementations of Scheme allow a program to redefine such a global variable (**car** for example) on condition that the redefinition modifies only this variable and not any other predefined function (not even those, like **map**, for example, that seem to use **car**). Only the functions explicitly in the program will see the new value of **car**. You can get this effect simply by **compute-kind**, but there is still a problem of knowing how to insert such a variable in the mutable environment. We could also invent a new special form, **redefine**, say, for this purpose. [see **Ex. 6.5**]

We'll add global variables to these environments by means of two functions, **g.current-extend!** and **g.init-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 ) )
```

The environments `g.current` and `g.init` return only addresses of variables, so we have to search for their values in the appropriate place. The containers where we search are simple vectors, values of the variables `sg.current` and `sg.init`. There's an initial `s` because these vectors represent memory, conventionally prefixed by `s` for store. Here[3] are the containers and the associated access functions. (However, we're not giving `predefined-update!` since it makes no sense for immutable variables.)

```
(define sg.current (make-vector 100))
(define sg.init (make-vector 100))
(define (global-fetch i)
  (vector-ref sg.current i) )
(define (global-update! i v)
  (vector-set! sg.current i v) )
(define (predefined-fetch i)
  (vector-ref sg.init i) )
```

To help define global environments, we'll provide two functions, `g.current-initialize!` and `g.init-initialize!`, to enrich (or modify) the static and dynamic environments synchronously.

```
(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 )
(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 )
```

3. For simplicity, we've limited the number of mutable global variables to 100, but that limitation will be lifted in Section 6.1.9.

Now we have an adequate arsenal to handle the pretreatment of variables and assignments. Those two have a similar structure: they analyze the classification returned by **compute-kind** and associate it with the correct access function. Notice that we don't use the functions **deep-fetch** nor **deep-update!** but the equivalent direct accessors when the variable we are searching for appears in the first activation record. Another clever trick (but one that some people would argue against) is that for predefined variables, we search for their value to be read right away (like a quotation) rather than at execution. In that way, we gain an access indexed to the vector of constant global variables.

```
(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) ) ) )
```

Assignment is similar in every way:

```
(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) ) ) )
```

Static Errors

The purpose of this pretreatment is so that such errors as an attempt to modify an immutable variable or to read a non-existing variable will be noticed during pretreatment rather than during execution. Those kinds of errors may even remain unnoticed if the erroneous forms are not evaluated. Such errors are signaled by the function **static-wrong** rather than by **wrong**, which we reserve for unforeseeable situations that occur during execution. The idea of a static error is useful but it clearly marks the difference between a free-handed language like Lisp and most others. If a program is valid in ML, then all its possible executions are exempt from type errors. Conversely, if we do not know how to prove that all evaluations of a program lead to errors, then we would have the tendency to think that the program is legal in Lisp. For example, consider the following definition:

```
(define (statically-strange n)
  (if (integer? n)
      (if (= n 0) 1 (* n (statically-strange (- n 1))))
      (cons) ) )
```

Even though it is statically wrong, this function provides a real service when applied to (positive!) integers. Whether we allow such a function or not depends on the spirit of the language; there's a compromise between the security we're looking for and the freedom we're ready to sacrifice for it. It is very important to be warned about errors as soon as possible—that's the position of ML—but, if we want no limits on our programming arsenal, if we want a little ease and a little taste of danger, then we'll prefer Lisp.

The function **static-wrong** should thus be understood as delivering a message about an anomaly but generating a result, valid for pretreatment; the pretreatment itself will raise the error if by chance its evaluation is needed. That is, the warning is tied to pretreatment; the error to execution. We make these ideas explicit in the way we define **static-wrong**.

```
(set! static-wrong
      (lambda (message . culprits)
        (display `(*static-error* ,message . ,culprits))(newline)
        (lambda (sr k)
          (apply wrong message culprits) ) ) )
```

In that way, we ascend to a nirvana for implementers where we can have our cake and eat it, too.

Remember that for mutable global variables, we have to verify that they've been initialized when we access them, in contrast both to local variables and to immutable global variables. Thus there is a cost for accessing mutable global variables. In the case of incremental compilation (as, for example, in a compiling interaction loop like `(display (compile-then-run (read)))`), we could slightly improve the pretreatment of mutable global variables that have already been initialized. In fact, that's what we did[4] earlier in `meaning-reference`. [see Ex. 7.6]

6.1.5 Starting the Interpreter

The interpreter reads an expression, pretreats it, and then evaluates it. In that way, it produces a compiling interaction loop.

```
(define r.init '())
(define sr.init '())
(define (chapter61-interpreter)
  (define (compile e) (meaning e r.init))
  (define (run c) (c sr.init display))
  (define (toplevel)
    (run (compile (read)))
    (toplevel) )
  (toplevel) )
```

However, before we start this interpreter, we must enrich its initial environment a little. We'll assume again that we have some macros available to hide the implementation details so that the following definitions will resemble what they've always been. Here are a few of them, to which we've added, of course, the indispensible `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 (+ arity 1)) )
    (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)
```

4. If it were possible to modify code in place, we could also imagine patching the instruction that verifies the initialization of a global variable so that it no longer does so if that's really the case. Bigloo interpreter [Ser94] adopted that solution.

```
            (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 Functions with Variable Arity

Our interpreter still lacks functions with variable arity. As always, those functions pose a few difficulties for us. As we have used them, activation records contain the values of arguments, but they also serve as the receptacles for bindings that will be created. In the case of functions with variable arity, the correspondence between these two effects is not reliable because there is no inevitable relation between the number of arguments passed to a function and its arity. For example, a function having `(a b . c)` as the list of variables could accept two, three, or more arguments without error, but it would bind only those three variables. For that reason, an activation record must always contain at least three fields. Simply put, for an application $(f\ a\ b)$, the activation record that's allocated must have a superfluous field (and that makes three fields in all) to authorize the invocation of any function capable of accepting at least two values, that is, those functions with a list of variables congruent to `(x y)` or `(x y . z)` or `(x . y)` or even `x`.

Functions with variable arity thus handle the activation record they receive in such a way as to put "excess" arguments into a list. The function `listify!` will be used for that purpose and indeed only for that purpose. Programming it is not complicated, but doing so obliges us to juggle various indices numbering the terms of the application, the variables to bind, and the fields of the activation record. The value of the variable `arity` represents the minimal number of arguments expected.

```
(define (meaning-dotted-abstraction n* n e+ r)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
         (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 ) ) ) ) )
```

Now we can pretreat all possible `lambda` forms by means of the following static analysis:

```
(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 Reducible Forms

Our interpreter could take advantage of a conventional way of improving compilers with respect to reducible forms, that is, applications where the function term is a `lambda` form. In such a case, there's no point in creating a closure to apply later; it's sufficient to assimilate the form to a block with local variables, in the style of Algol. By the way, `((lambda ... ... ) ... )` is nothing other than a disguised `let`; it opens a block furnished with local variables. The case of functions with fixed arity is thus simplicity itself, but once again[5] that's not so for functions with variable arity. Not providing the right number of arguments to a function is now a static error that can be raised in pretreatment.

```
(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 less arguments" e ee*) ) )
            ((null? n*)
             (if (null? e*)
                 (meaning-fix-closed-application
                  nn* (cddr e) ee* r )
                 (static-wrong "Too much 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)) ) ) ) )
```

5. You can see by now why so many languages do not support functions with variable arity in spite of their usefulness.

For functions with variable arity, we can avoid using **listify!** since here the arity of the function and the number of arguments are both known statically. The solution uses a variation on **meaning*.** Here we call that variation **meaning-dotted*.** It behaves like **meaning*** for obligatory arguments, but it builds a list of "excess" arguments on the fly by inserting the necessary calls to **cons.** Doing that entails a lot of code for a case that's fairly rare. However, we must explicitly use **()** to initialize the superfluous field in the activation records; the function **meaning-no-dotted-argument** does that. All that comes down to making a change on the fly, like this:

```
((lambda (a b . c) ...)          ≡   ((lambda (a b c) ...)
 α β γ δ ...)                         α β (cons γ (cons δ ...)) )
```

So here are those functions:

```
(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))
        (rank (- size (+ (length e*) 1))) )
    (if (< rank arity)
        (lambda (sr k)
          (m sr (lambda (v)
                  (m* sr (lambda (v*)
                           (set-activation-frame-argument! v* rank 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 (+ arity 1)))
    (lambda (sr k)
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        (k v*) ) ) ) )
```

6.1.8 Integrating Primitives

We can gain efficiency from another important source by cleverly pretreating calls to the predefined functions of the immutable global environment. An application, such as `(car` α`)`, currently imposes the following incredible and painful sequence of steps:

1. Dereference the global variable `car`.
2. Evaluate α.
3. Allocate an activation record with two fields.
4. Fill that first field with the value of α.
5. Verify that the value of `car` really is a function.
6. Verify whether the value of `car` actually accepts one argument.
7. Apply the value of `car` to the argument. This step leads additionally to testing whether the argument really is a dotted pair for which it is legal to take the `car`.

We eliminate several of those steps if we're working in a strongly typed language, and that's what makes such languages so fast. In contrast, some of these verifications can be eliminated in a language like Lisp by pretreatments if such things can be verified statically. Since `car` is a global variable that cannot be modified, we can verify beforehand that it's a function that accepts one argument. In that way, we save step 5 (verify whether it's a function) and step 6 (verifying its arity). We can save even more by not allocating the activation record (steps 3 and 4), but inserting the code itself into the primitive being called (step 1).

This kind of integration—calling the primitive directly without going through a complete and consequently burdensome protocol—is known as *inlining*. In such circumstances, the only remaining steps are 2 and 7. A good compiler could still save a little in step 7 by factoring type tests so they would never be duplicated. For example, in the program `(if (pair? e) (car e) ... )` there is no point in `car` verifying whether its argument is a dotted pair because that is obviously and surely true. One method for doing so anyway is to consider `(car` x`)` as a macro equivalent to `(let ((`v x`))(if (pair?` v`) (unsafe-car` v`) (error ... )))` where `unsafe-car`[6] extracts the `car` of its argument if that argument is a pair but leads to unforeseeable side effects when that is not the case. All we have to do is to transform the code in order to migrate type-checks as far upstream as possible and to eliminate redundant tests. Here again, there is no problem in migrating computations because type-checks are immediate computations without possible errors.

A real compiler does not access values of immutable global variables since such variables belong to the realm of execution rather than to pretreatment. Besides, we don't have much need of these values; we only need to know whether they are functions and whether their arity is compatible with the function that we are

6. Primitives analogous to `unsafe-car` exist in most implementations in order to serve as targets for transformations of programs. These transformations should guarantee that `unsafe-car` is always used in safe contexts. In contrast, it is essential for an efficient compiler to be able to resort to these shortcut primitives and thus get rid of useless type-checking.

trying to pretreat. We'll add a new environment. Its role will be to describe the value of immutable global variables. That environment will be named **desc.init**, and it will associate a descriptor with a variable whose value is a function. The descriptor of a function will be a list; the first term of that list will be the symbol **function**; the second term will be the "address" of the primitive (which must really be invoked); the succeeding terms indicate the arity. We'll put the management of these descriptions inside the macro **defprimitive**, accompanied here by only one of its submacros, **defprimitive3**, to give you an idea of its siblings.

```
(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!                    ;** Modified **
          'name '(function ,value a b c))
         behavior ) ) ) ) )
```

The functions to manage this environment look like this:

```
(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)) ) )
```

We can explicate completely how applications are pretreated; that is, how they are analyzed so that they can be handed off to the right pretreatment. Notice that if a primitive appears in an application with the wrong arity, that anomaly will be indicated statically.

```
(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 for" e) ) )
                ) ) ) ))
   ((and (pair? e)
         (eq? 'lambda (car e)) )
    (meaning-closed-application e e* r) )
   (else (meaning-regular-application e e* r)) ) )
```

Applications implicating known primitive functions are handled like this:

```
(define (meaning-primitive-application e e* r)
  (let* ((desc (get-description e)) ;desc = (function address . variables-list)
         (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)) ) ) )
```

The preceding integration involves only primitives with fixed arity. Primitives with variable arity, like **append**, **for-each**, **list**, **map**, *, +, and several others (except **apply**) in general can be considered as macros for which the expansion reduces to cases we've already studied. For example, (**append** π_1 π_2 π_3) can be rewritten as (**append** π_1 (**append** π_2 π_3)). That transformation lets us integrate forms with variable arity but does not imply that these functions have fixed arity. (**apply append** π) is an example where **append** will be called with a variable arity.

We have integrated only calls with three or fewer arguments. The reason for this limitation is simple: in Scheme, there are no essential functions of fixed arity that have more than three arguments anyway!

6.1.9 Variations on Environments

Accessing deep local variables (that is, those that do not belong to the first activation record) have a linearly increasing cost because we have to run through the linked records to find them. There is a simple technique—known as *display*—to access such variables in constant time. To do so, every deep activation record has to be accessible from the first one, as in Figure 6.3. In that way, every deep variable can be read or written by one indirection (to determine which record) and one offset (inside that record). However, even if accessing deep variables is faster in this way, the cost of allocating linked records is greater than before because every activation record must refer to all the deep records. Of course, it's possible to set up only the links really used, but doing so requires analyzing which variables are consulted. Additionally, this technique ruins our interpreter since with it, we will no longer know how large an activation record to create before the function to invoke checks the depth of its closed environment. For those reasons, we have to allocate the *display* somewhere else or even limit the maximal authorized depth (though such a limit is not very Lispian).

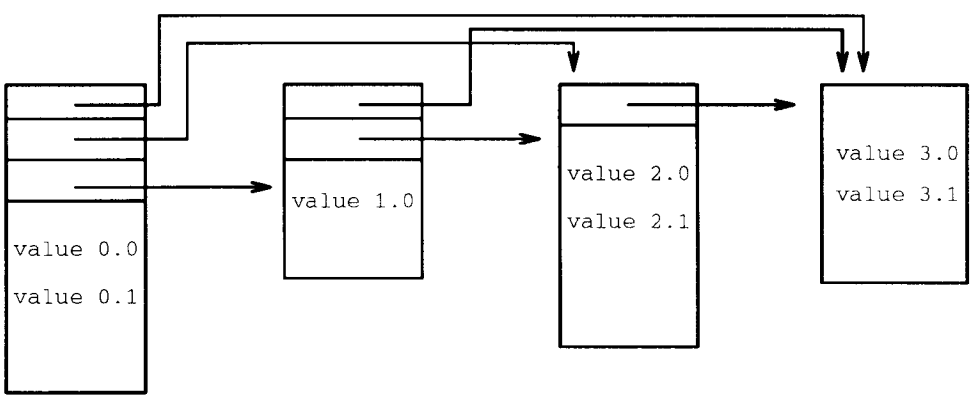


Figure 6.3 *Display* technique

Flat Environment

Another technique would be to adopt flat environments. When a closure is created, instead of closing the entire environment, we could build a new environment containing only the variables that are really closed. The cost of building the closures is thus higher, but in contrast the environment has at most only two activation records: the one that contains arguments and the one that contains closed variables. Then in order to make sure we can still share variables if need be, it's a good idea to transform things by a box. [see p. 115]

Finally, it's feasible to mix all these possibilities in order to adapt them to the cases where they excell. To carry out this adaptation, we must analyze programs more precisely. In short, we need a real compiler, not just an interpreter with pretreatments like ours.

Defining Global Variables

When it encounters a global variable, the current interpreter recognizes it by the fact that it belongs to the global predefined or mutable environment. Consequently, we must declare all the global variables that we're going to use. We do so in the definition of the interpreter by means of the macro **defvariable**:

```
(define-syntax defvariable
  (syntax-rules ()
    ((defvariable name) (g.current-initialize! 'name)) ) )
```

Regardless of the practices for declaring variables in other languages, this way of doing things is considered quite out of place in Lisp dialects. Consequently, when an identifier is used as a global variable, we want it to be created automatically. One easy solution is to improve the function **compute-kind** so that it detects such cases, like this:

```
(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)) ) )
```

The problem is that in doing so, we've slightly violated the pretreatment discipline that we've adopted. In effect, the global variable is created during pretreatment rather than during execution. Let's analyze what goes on before the process of creating a variable. When a new variable is taken into account, two different results occur:

1. Its name is added to the environment of mutable global variables (known as **g.current**) and in passing, a number is assigned to it.
2. An address is allocated to the variable; the address contains its value, conforming to the number assigned to it.

In that light, consider the following program:

```
(begin (define foo (lambda () (bar)))
       (define bar (lambda () (if #t 33 hux))) )
```

Pretreating that program turns up three new variables: **foo**, **bar**, and **hux**. As soon as they are encountered, these variables are added by pretreatment to the global environment and receive a number. So after pretreatment, would you say that they have been created or merely exist potentially? Would you say that the variable **foo** appearing in the **define** form was only semi-created, or that **hux** is only semi-semi-created?

To execute a pretreated program, we assume that the mutable global execution environment contains the values of these new variables. Consequently, we must extend it at just the right moment: the beginning of the execution phase is, in fact, the only possible instant. Before the execution of a pretreated program, the size of the global execution environment is adapted to the number of mutable global variables present. The function **stand-alone-producer** illustrates that idea. It pretreats a program in a closure **(lambda (sr k) ...)**, and the first act of that closure is to allocate the *ad hoc* global environment.

```
(define (stand-alone-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) ) ) )
```

There may be predefined variables in the mutable global environment so that environment can serve as the communication point between the system and a program, as for example the base used to read or write numbers. For that reason, we'll assume that these variables appear in the environment returned by **(original.g.current)**.

With this style of programming, all variables exist as soon as they are mentioned. In a way closer to Scheme, we can also allow variables to be introduced only by the form **define**. In this latter case, and for the previous example, pretreatment must indicate a static error. There, **hux** is an undefined variable even it it is neither read nor written. That's not the case of **bar**: it isn't defined at its first use, but it is defined later.

To show the details of how **define** works, we'll add it as a new special form to the **meaning** pretreater:

```
... ((define) (meaning-define (cadr e) (caddr e) r)) ...
```

When definitions are analyzed, there is a test to see whether the variable exists already; if it does, the definition behaves like an assignment.

```
(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) )
```

Now the program pretreater has the responsibility of verifying whether any global variables have suddenly appeared without yet being explicitly defined. The list of defined global variables is thus put together at the beginning of pretreatment.

```
(define (stand-alone-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 "Not explicitely defined 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)) )
      '() ) )
```

In summary, we must distinguish what's within the jurisdiction of pretreatment and what belongs to execution. A definition at execution is rigorously like assignment. In contrast, its pretreatment induces verifications, either instantaneous or deferred until the end of pretreatment. The usual special forms (other than **define**) have an effect only at execution. They have dynamic semantics. Definition belongs to static semantics (implemented by pretreatment) since it will not be aware of any global variable that is not accompanied by a definition. Moreover, it's a static error to violate this rule.

As we said before about local variables of **letrec** [see p. 60], for variables, we must distinguish the idea of existence from initialization. The fact that a variable exists does not mean that it has been initialized. Here's an example to clarify that point:

```
(begin (define foo bar)
       (define bar (lambda () 33)) )
```

In the case of a compiling interaction loop (a kind of incremental compiler immediately evaluating whatever it just compiled), all these problems are simplified because the end of pretreatment coincides with the beginning of execution of pretreated code, and these two phases occur in the same memory space. Thus we can adopt the first variation of **adjoin-global-variable!** as well as the present improvement in **meaning-reference** which consisted of suppressing the test about initialization for variables that have already been initialized.

6.1.10 Conclusions: Interpreter with Migrated Computations

It's hard to measure the improvements of this interpreter, but the gain is on the order of 10 to 50. Pretreatment is fast enough even if still improvable (mainly that **compute-kind** may use hashtables rather than lists for environments). In addition to these non-negligible gains, we've also taken advantage of the ideas of static computations (what is pretreated) and dynamic computation (what is left until execution). In a certain way, then, the role of a good compiler is to leave the minimal amount of work to do at execution. To invent the best pretreatments, there have been many analyses carried out to identify which properties are valid at execution. We'll mention only a few: abstract interpretation in [CC77], partial evaluation in

[JGS93], and flow control analysis in [Shi91]. Another area for improvement is how to choose good data structures, as you can see from the discussion about activation records.

Pretreatment reveals certain errors sooner and independently of their possible execution. It paves the way for a safer and more efficient programming style since fewer verifications are left until execution. The kind of pretreatment we looked at here is quite rudimentary, but it could certainly be extended to check types, as in ML.

The interpreter we finally got strongly resembles the one from Chapter 3 except that we've replaced closures by objects. Even though closures and objects have very similar internal representations, closures are poor man's objects: they are opaque; they can only be invoked; and we don't even know how to confer new behavior on them, for example, to introduce a little reflection.

6.2 Rejecting the Environment

Every expression is evaluated in a unique residual environment; its value is `sr`. Since that environment changes only during the invocation of functions that reinstall and then extend their definition environment, you might ask whether it would be more useful to introduce the idea of the current environment, the value of which is the global variable (or even the register) `*env*`. Making that environment global lets us avoid explicitly passing environments as arguments to all the closures resulting from pretreatment. In other words, the result of pretreatment will no longer be an abstraction like `(lambda (sr k) ... )` but rather simply `(lambda (k) ... )`.

To carry out this transformation, which will suppress local variables `sr` in favor of one global variable `*env*`, all we have to modify is the function `meaning` to introduce management of this variable there. *A priori* the modification seems simple:

1. We search for local variables in `*env*` now, not in `sr` anymore.
2. When a function is invoked, it assigns its own definition environment, conveniently extended already, to `*env*`.

However, on closer examination, we see that those steps are not sufficient because if `*env*` is assigned, then from time to time it will be necessary to restore its earlier value, if only to return to a computation that was interrupted during an invocation. Here's a trivial solution: when a function assigns an environment to `*env*`, the function stores the preceding value and restores that value at the moment that the function returns its final value. The problem with this solution is that it does not conform to Scheme, which demands that tail calls must be evaluated with a constant continuation.

This problem resembles the well known problem of defining a function-calling protocol in terms of machine registers. The registers have to be saved during the computation of the invoked function, but who should do it?

- The one being called knows exactly which registers it uses and as a consequence, it can limit its efforts to preserving only those that concern it. The

difficulty here is that the one being called then begins by saving registers, evaluating its body, saving the value produced, restoring the registers, and then returning the final value. The body of the called function is thus evaluated with a continuation which is that of the caller plus that fact that the registers must be restored.

- The caller knows exactly which registers it will need after the invocation and can thus save just those itself. Then the one being called has only to evaluate its body and return the value produced. If the caller does not have to restore the registers itself, you see that the one being called does not add any constraints. However, this technique is obviously too punctilious since the one being called may need only a few registers and could get along with registers not used by the caller without requiring the caller to save anything at all.

[SS80] proposed the caller should mark registers in one of two ways: "must be saved if used" or "can be overwritten without harm." The one being called can then save only what is really needed and change the mark to "must be restored" or "has not changed." This technique boils down to making each register a stack where only the top is accessible and the depths store values that must be restored. During a return, a special machine instruction restores the registers according to their marks.

It is very important for tail calls to be executed with a constant continuation (that is, without increasing the size of the recursion stack) so that iterations can be efficient and not hampered by the size of the recursion stack. For those reasons, we'll adopt the second strategy, so the caller will be responsible for preserving the environment.

Fortunately, whether or not an evaluation is in tail position is a static property, so we will add a supplementary argument to the function **meaning**. That supplementary argument is a Boolean, `tail?`, indicating whether or not the expression is in the tail position. If the expression is in the tail position, it is not necessary to restore the environment. (It's superfluous to save that environment, but it is not forbidden to do so.)

How do we determine the expressions in tail position? It's sufficient to look at the denotations of Scheme: every subform evaluated with a continuation different from the continuation of the form containing it is not in tail position. In an assignment (`set!` x π), the subform π has a continuation different from that of the assignment since its value must be written in the variable x. Therefore, it is not in tail position. Likewise, in a conditional (`if` π_0 π_1 π_2), the condition π_0 is not in tail position. In a sequence (`begin` π_0 ... π_{n-1} π_n), the forms $\pi_0 \ldots \pi_{n-1}$ are not in tail position so we must save the environment between the computation of various terms of the sequence. In an application (π_0 ... π_n), none of the terms are in tail position since the invocation still remains to be done. In contrast, the body of a function is in tail position as well as the evaluation of the entire program since neither the one nor the other will need to restore the previous environment.

Here is a new version of the function **meaning** followed by the function that starts this new interpreter.

```
(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 (chapter62-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    ((meaning (read) r.init #t) display)
    (toplevel) )
  (toplevel) )
```

6.2.1 References to Variables

A reference to a variable now uses the register ***env***. We won't give a new version of **meaning-assignment** since you can deduce it easily enough.

```
(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 Alternatives

We'll skip over quoting; it only has to be (or not be) in tail position. We'll go on to the conditional. Since the condition is not in tail position, you see this:

```
(define (meaning-alternative e1 e2 e3 r tail?)
  (let ((m1 (meaning e1 r #f))          ;restore environment!
        (m2 (meaning e2 r tail?))
        (m3 (meaning e3 r tail?)) )
    (lambda (k)
      (m1 (lambda (v)
            ((if v m2 m3) k) )) ) ) )
```

6.2.3 Sequence

The last expression of a sequence saves the environment if the sequence must do so. Notice that the current environment is restored only if there have been applications that might have modified it. In particular, a sequence like `(begin a (car x) ...)` does not require that the environment be preserved during the computation of `a` nor `(car x)` because it won't be modified.

```
(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?)) )
    (lambda (k)
      (m1 (lambda (v)
            (m+ k) )) ) ) )
```

6.2.4 Abstraction

An abstraction must capture the current environment, which is the "birth" environment of the closure being created. The abstraction must restore the environment to extend it to other invocation instances. The case of functions with variable arity is similar to that of functions with fixed arity.

```
(define (meaning-fix-abstraction n* e+ r tail?)
  (let* ((arity (length n*))
         (arity+1 (+ arity 1))
         (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 Applications

The only really complicated case is that of an application since an application has to manage whether or not the environment must be restored after the invocation.

```
(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) ))
                   (wrong "Not a function" f) ) )) )
        (lambda (k)
          (m (lambda (f)
               (if (procedure? f)
                   (m* (lambda (v*)
                         (let ((sr *env*))          ;save environment
                           (f v* (lambda (v)
                                   (set! *env* sr) ;restore environment
                                   (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-argument r size tail?) ) )
(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (rank (- size (+ (length e*) 1))) )
    (lambda (k)
      (m (lambda (v)
           (m* (lambda (v*)
                 (set-activation-frame-argument! v* rank v)
                 (k v*) ) )) )) ) )
(define (meaning-no-argument r size tail?)
  (let ((size+1 (+ size 1)))
    (lambda (k)
      (let ((v* (allocate-activation-frame size+1)))
        (k v*) ) ) ) )
```

In this way, we've produced a new interpreter with the environment put into a

register. The initial global environments are the same as before.

6.2.6 Conclusions: Interpreter with Environment in a Register

This transformation is not always helpful. If we want to add parallelism to the language, the global variable ***env*** would be a unique resource shared by all tasks. Moreover it would have to be saved in the context of every task. However, the fact that the environment is always available is advantageous if we want to add reflection to the language because we can then imagine primitives accessing it.

This new interpreter is no faster than the preceding one. Of course invocations now have only one variable instead of two, but they do so at the expense of a reference to a global variable that can change every time the environment has to be checked. On the positive side, the idea of tail position is clearer now, and we are gently getting closer to the next interpreter.

6.3 Diluting Continuations

It is not rare for the implementation language to provide sorts of continuations that we can use directly (rather than handle them explicitly as in the preceding interpreter). That situation is not as crazy as it sounds. If we have a compiler available for Scheme, we usually get an interpreter for it by writing one in Scheme and compiling it. The interpreter we get that way then uses the execution library, especially the `call/cc` it finds there. However, if we compile only Lisp, the only continuations we'll ever need are equivalent to **setjmp/longjmp** from the C language library. In these two cases, `call/cc` can be considered as a magic operator, and it is therefore totally pointless to reify continuations everywhere. Always having them on hand requires an extraordinary rate of allocation, allocations that we give up if we are trying to increase the interpretation speed.

The next interpreter will thus return to a direct style without explicit continuations. We'll take advantage of it to make two new modifications:

1. functions will now be represented explicitly as *ad hoc* objects;
2. the results of pretreatment will appear as combinators (written as capital letters) reminding us of the instructions of a hypothetical virtual machine.

We'll take up all the improvements in the first interpreter (calls to primitives, reducible forms, etc.) again and then give them all definitions again.

6.3.1 Closures

As usual, closures will be represented by objects with two fields, one for their code and another for their definition environment. The invocation protocol will be adapted to this representation by the function **invoke**, like this:

```
(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) ) )
```

The code of interpreted closures will thus be represented by a closure with two variables, one for the activation record, and the other for the definition environment. In that way, the definition environment will be available for extensions. Every invocation of a closure must thus pass by the invocation function, named (reasonably enough) **invoke**.

6.3.2 The Pretreater

Pretreatment of programs is handled by the function **meaning**. Instead of returning a closure `(lambda (k) ... )`, now it returns `(lambda () ... )`, an object that we can interpret as a address to which we simply jump to execute it. (This practice descends directly from Forth.) By suppressing all variables, we get thunks. Moreover, it won't be hard to have an invocation protocol slightly more elaborate than the current one to produce a simple but effective `GOTO` **[see Ex. 6.6]** to invoke thunks.

You can see the function **meaning** on page 207.

6.3.3 Quoting

Quoting is still quoting, but it appears even easier to read because of the combinator `CONSTANT`, like this:

```
(define (meaning-quotation v r tail?)
  (CONSTANT v) )
(define (CONSTANT value)
  (lambda () value) )
```

6.3.4 References

Pretreating variables always involves categorizing them and then associating them with the right reader. These readers will be generated by appropriate combinators. Since we use combinators, this definition is lighter and consequently easier to read.

```
(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 ) ) ) )
```

Nevertheless, notice that in the combinator **CHECKED-GLOBAL-REF**, when the variable is not initialized, since we have available only the index of the variable in the vector **sg.current**, it is no longer possible simply to indicate the name of the erroneous variable. To do that, we have to keep what we conventionally call a "symbol table" (here, the list **g.current**) indicating the names of variables and the locations where they are stored. [**see Ex. 6.1**] With that device, if we know the index of a faulty variable, then we can retrieve its name.

6.3.5 Conditional

Conditionals are clearer here, too, because of the combinator **ALTERNATIVE**, which takes as arguments the results of the pretreatment of its three subforms.

```
(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 Assignment

The structure of assignment resembles the structure of referencing except that there is a subform to evaluate, a subform provided as an argument to all the write-combinators.

```
(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 Sequence

We express a sequence clearly in terms of the combinator **SEQUENCE** corresponding to a binary **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 Abstraction

Closures are created by the combinators **FIX-CLOSURE** or **NARY-CLOSURE**. Their differences involve verifying the arity and putting the excess arguments into a list.

```
(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 (+ arity 1)))
    (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 (+ arity 1)))
    (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 Application

The only thing left to handle is applications. **meaning-application** analyzes their nature. It recognizes reducible forms, calls to primitives, and any applications.

```
(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?)) ) )
```

All applications are subject to only four combinators. In the combinator TR-REGULAR-CALL, the computations of the function term and the arguments are put into sequence. As usual, the evaluation order is left to right.

```
(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-argument r size tail?) ) )
(define (meaning-some-arguments e e* r size tail?)
  (let ((m (meaning e r #f))
        (m* (meaning* e* r size tail?))
        (rank (- size (+ (length e*) 1))) )
    (STORE-ARGUMENT m m* rank) ) )
(define (meaning-no-argument 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* rank)
  (lambda ()
    (let* ((v (m))
           (v* (m*)) )
      (set-activation-frame-argument! v* rank v)
      v* ) ) )
(define (ALLOCATE-FRAME size)
  (let ((size+1 (+ size 1)))
    (lambda ()
      (allocate-activation-frame size+1) ) ) )
```

6.3.10 Reducible Forms

Since reducible forms include an explicit **lambda** form in the function position, their pretreatment adds four new combinators. **CONS-ARGUMENT** creates the list of excess

arguments. **ALLOCATE-DOTTED-FRAME** creates an activation record, very much like **ALLOCATE-FRAME** except that the supplementary field is explicitly initialized by () (an initialization we avoided in **ALLOCATE-FRAME** for performance reasons).

```
(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-argument 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?))
        (rank (- size (+ (length e*) 1))) )
    (if (< rank arity) (STORE-ARGUMENT m m* rank)
        (CONS-ARGUMENT m m* arity) ) ) )
(define (meaning-no-dotted-argument 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 (+ arity 1)))
    (lambda ()
      (let ((v* (allocate-activation-frame arity+1)))
        (set-activation-frame-argument! v* arity '())
        v* ) ) ) )
```

The combinator **FIX-LET** must restore the current environment—implying that it must have been stored somewhere earlier so that it could be restored eventually. There is an elegant solution here: since it appears in the linked activation records, we simply have to go look for it there.

6.3.11 Calling Primitives

The last type (and not the least important in number) is the case of forms where we have an immutable global variable in the function position. In that case, we will call the right invoker, using the arity as a parameter, so that we no longer need to re-verify the arity.

```
(define (meaning-primitive-application e e* r tail?)
  (let* ((desc (get-description e))
         ;;desc = (function address . variables-list)
         (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)) )
               (address v1 v2 (m3)) )) )
```

CALL3 explicitly puts the arguments in sequence to respect the left to right order.

6.3.12 Starting the Interpreter

Since continuations are no longer explicit in this interpreter, we must look again at the macros that enrich the global environment. Their structure has changed little, so we'll show you only **defprimitive2** by way of example:

```
(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) ) ) ) ) )
```

We start the interpreter by this:

```
(define (chapter63-interpreter)
  (define (toplevel)
    (set! *env* sr.init)
    (display ((meaning (read) r.init #t)))
    (toplevel) )
  (toplevel) )
```

6.3.13 The Function call/cc

Now since the function `call/cc` is magic, for its own definition, it needs the function `call/cc` from the library on which the interpreter is built, so here we have the tautologic definitions of the first interpreters.

```
(definitial call/cc
  (let* ((arity 1)
         (arity+1 (+ arity 1)) )
    (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 The Function apply

To look beyond our usual horizon, here's the function **apply**. It is always difficult to write because it strongly depends on how functions themselves are coded and on which calling protocol has been chosen, but here it is in all its detail and complexity.

```
(definitial apply
  (let* ((arity 2)
```

```
      (arity+1 (+ arity 1)) )
  (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 ) ) )
```

That primitive verifies that it has at least two arguments and then runs through the activation record to find the exact number of arguments to which the function in the first argument applies. To do so, it must compute the length of the list in the last argument. Once it has this number, we can then allocate an activation record of the correct size. We copy into that record all the arguments in their correct position; the first ones we copy directly from the activation record furnished to `apply`; the following ones, by exploring the list of "superfluous" arguments. That exploration stops when that list is empty, as determined by the test `null?`. It might seem more robust to use `atom?` but that would make the program `(apply list '(a b . c))` correct—contrary to the norms of Scheme.

As you can see, `apply` is not an inexpensive operation because it allocates a new activation record, and it must run through the list in the last argument.

6.3.15 Conclusions: Interpreter without Continuations

This new interpreter is two to four times faster than the previous one, mainly because we don't reify continuations in it. In effect, representing continuations by closures mixes them up with other, more ordinary values. Doing that disregards an important property of continuations: that they habitually have a very short extent. For that reason, allocating continuations on a stack is usually a winner because it is a low-cost strategy. That's also the case for de-allocations, usually just one instruction popping the pointer from the top of the stack. Just so we don't make a mistake here, we should repeat that the mass of data allocated to represent a continuation is the same order of magnitude whether on a stack or in a heap, but managing it on a stack is less costly than managing it in a heap. (See also [App87] for a different opinion that does not take locality into account.) This observation

holds even if we imagine specializing the heap in several zones with one adapted to continuations, as described in [MB93].

The thunks that this most recent interpreter uses belong to the technique of *threaded code* as in [Bel73] and common in Forth [Hon93]. This interpreter is also strongly inspired by that of Marc Feeley and Guy Lapalme in [FL87].

Combinators actually play the role of code generators. We introduced them in this interpreter because they offer a simple interface that makes it easy to change the representation of pretreated programs. With them, we can easily imagine building objects rather than closures. In fact, there's an exercise [**see Ex. 6.3**] in preparation for the next chapter, where we'll see their use in compilation.

6.4 Conclusions

At first glance, this chapter and its three interpreters might seem like a giant step backward, wiping out all the progress we had made in the first four chapters. Indeed, we have practically suppressed memory (except for managing activation records), and continuations have disappeared. On the positive side, we've presented the idea of pretreatment as a preliminary to compilation. We've also separated static from dynamic and made various improvements to increase execution speed. We actually achieved that last goal: we've improved speed by roughly two orders of magnitude in comparison with the denotational interpreter.

The third interpreter of this chapter actually implements all the special forms of Scheme and represents the essence of any language. All that's left to do is endow the interpreter with a memory manager and *ad hoc* libraries specialized for editing text, industrial drafting and design, Scheme as a language, materials testing, virtual reality, etc. Of course, when we say "all that's left," we're glossing over the incredible complexity of choosing representation schema for primitive objects in memory where the chief goals are rapid type checking as in [Gud93] and no less efficient garbage collection.

Of course, we could improve these interpreters or even extend them to handle new special forms, but remember that our real purpose is to produce a set of interpreters modified incrementally to reduce the mass of detail composing them and to highlight new goals that they illustrate.

6.5 Exercises

Exercise 6.1 : [see p. 213] Modify the combinator `CHECKED-GLOBAL-REF` so that the error message about an uninitialized variable is more meaningful.

Exercise 6.2 : Define the primitive `list` for the third interpreter of this chapter. You will, of course, do so elegantly and with little effort.

Exercise 6.3 : Instead of pretreating a program, we could display the way it would be pretreated. Here's what we mean for factorial:

```
? (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 )
```

Write such a disassembler to display pretreatment.

Exercise 6.4 : Modify the last interpreter of this chapter so that activation records are allocated before arguments are computed. Then arguments could be put directly into the right place. **[see p. 189]**

Exercise 6.5 : Define a special form, `redefine`, to take a variable in the immutable global environment and insert it in the mutable global environment, as we discussed on page 191. The initial value of this new global variable is the value that it had before.

Exercise 6.6 : Improve the pretreatment of functions without variables. **[see p. 212]**

Recommended Reading

The last interpreter in this chapter is inspired by [FL87]. The method of getting from the denotational interpreter to combinators comes from [Cli84]. If you really like fast interpretation, you'll enjoy meditating on [Cha80] and [SJ93].

7
Compilation

THE preceding chapter explicated a pretreatment procedure that transcribed a program written in Scheme into a tree-like language of about twenty-five instructions. This chapter exploits the results of that pretreatment to transform it into a set of bytes, an *ad hoc* machine language. We'll look at each of the following ideas in turn: defining a virtual machine, compiling into its own language, and implementing various extensions of Scheme, such as escapes, dynamic variables, and exceptions.

(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 *address*)
(CALL1 *address m*1)	(CALL2 *address m*1 *m*2)
(CALL3 *address m*1 *m*2 *m*3)	(FIX-CLOSURE *m*+ *arity*)
(NARY-CLOSURE *m*+ *arity*)	(TR-REGULAR-CALL *m m*∗)
(REGULAR-CALL *m m*∗)	(STORE-ARGUMENT *m m*∗ *rank*)
(CONS-ARGUMENT *m m*∗ *arity*)	(ALLOCATE-FRAME *size*)
(ALLOCATE-DOTTED-FRAME *arity*)	

Table 7.1 The intermediate language with 25 instructions: m, $m1$, $m2$, $m3$, $m+$, and v are values; $m*$ returns an activation record; *rank*, *arity*, *size*, i, and j are natural numbers (positive integers); *address* represents a predefined function (subr) that takes values and returns one of them.

Compilation often produces a set of fairly low-level instructions. That was not the case in the pretreatment from the previous chapter. In fact, it built the equivalent of a structured tree. For a more eloquent example, consider the following program:

```
((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)))) )) )
```

After transcription, its pretreatment looks like this:

```
(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 ) ) )
```

It's not easy to read, but it's accurate. The purpose of this chapter is to show that this form is far from final. Indeed, certain transformations, such as linearizing and byte-coding, can even transmute it into other languages. The language of the twenty-five instructions/generators in Table 7.1 will serve as our intermediate language, a kind of springboard for leaping into other realms.

We'll regard the pretreatment phase (that last interpreter in the preceding chapter, the one that produced the famous intermediate language) as the first pass of a compiler. Consequently, we'll be interested only in the twenty-five functions/generators. In fact, we'll adapt them to the characteristics of our final target language. By dividing the work in this way, we take advantage of the fact that the intermediate language is executable. We've already used that fact to test the pretreater. Now it will let us concentrate solely on the second pass.

First, we'll study compilation toward a virtual machine. It will be a simple one, but one that presents all the characteristics of any machine programmable in machine language. Its instruction set will be represented by bytes, in particular, integers from 0 to 255. This technique is known as *byte-coding*. It appeared sometime before 1980, according to [Deu80, Row90]. Since then, it has often been used to highlight the rudiments of compilation, as in [Hen80]. The code we get this way is particularly compact, a quality that justifies its use on machines with little memory or limited caching. It's the technique used by PC-Scheme [BJ86] and Caml Light [LW93].

There are several aspects to the entire technique. After pretreatment, a program is compiled into byte-code vectors. These byte-codes are then evaluated by an interpreter. That is, compilation and interpretation are both involved, but only interpreting byte-code is necessary for execution.

7.1 Compiling into Bytes

Our goal is little by little to bring up a machine specialized to interpret byte-code. We define this machine by defining the twenty-five instructions of the intermediate language. Some of these definitions are obvious, but others will require a little inventiveness on our part. To our advantage, we'll be designing both the machine and its instruction set at the same time. This flexibility will be indispensible when we want, say, to add new registers or introduce a stack.

To get the ultimate byte-code vector, we'll have to linearize the program expressed in intermediate language. For that reason, we must be sure that communication between instructions is limited to the common resources of the machine, that is, the registers and the stack. For the moment, our machine has only one register; it contains the current lexical environment, `*env*`, but we'll soon fatten up this somewhat Spartan architecture.

7.1.1 Introducing the Register *val*

Among the twenty-five instructions in the intermediate language, some of them produce values, for example, like the instructions `SHALLOW-ARGUMENT-REF` or `CONSTANT`, whereas others, like `FIX-LET` or `ALTERNATIVE`, coordinate computations. In that light, let's look more closely at `GLOBAL-SET!`. It was defined like this:

```
(define (GLOBAL-SET! i m)
  (lambda () (global-update! i (m))) )
```

To break communication, we have to have another register. We'll call it `*val*`. Instructions that produce values will put them there so that consumers can find them. Consequently, an instruction like `CONSTANT`—one that produces values—will be written like this:

```
(define (CONSTANT value)
  (lambda ()
    (set! *val* value) ) )
```

In contrast, a consumer of values, like `GLOBAL-SET!`, will become:

```
(define (GLOBAL-SET! i m)
```

```
(lambda ()
  (m)
  (global-update! i *val*) ) )
```

The form `(m)` will eventually produce a value in the register `*val*`, which will then be transferred by `global-update!` to the right address. Notice that `global-update!` does not disturb the register `*val*`. To disturb it would cost at least one instruction. As a consequence, the value of a form that assigns something to a global variable is the value found in the register `*val*`, that is, the assigned value.

It's easy enough to deduce the rest of the transformation of the two examples we gave earlier. For example, we linearize `SEQUENCE` automatically, like this:

```
(define (SEQUENCE m m+)
  (lambda () (m) (m+)) )
```

while `FIX-LET` becomes this:

```
(define (FIX-LET m* m+)
  (lambda ()
    (m*)
    (set! *env* (sr-extend* *env* *val*))
    (m+)
    (set! *env* (activation-frame-next *env*)) ) )
```

7.1.2 Inventing the Stack

We've already achieved part of the linearizing that we wanted to do, but for some instructions, the issues are more subtle. For example, `STORE-ARGUMENT` has become:

```
(define (STORE-ARGUMENT m m* rank)
  (lambda ()
    (m)
    (let ((v *val*))
      (m*)
      (set-activation-frame-argument! *val* rank v) ) ) )
```

That instruction uses the form `let` to save values and restore them later if needed. Here, the form `let` saves a value in an "anonymous register" `v` while `(m*)` is being calculated. We can't associate a real machine register with `v` because we might need more than one such `v` simultaneously, especially in the case of multiple forms of `STORE-ARGUMENT` nested inside the computation of `(m*)`. Consequently, we need a place where we can save any number of values. A stack would be useful here, the more so because the pushes and pops seem equally balanced. We'll assume then that we have a well defined stack managed by the following functions:

```
(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*) )
```

Endowed with this new technology, we can adapt the instruction **STORE-ARGUMENT** to indulge immoderately in pushing and popping the stack. Instead of saving values in a register, we'll keep them on a stack, and we'll get them back from the stack when needed as well. This plan works only if we insure that the stack that **(m*)** takes is the same one that **(m*)** returns. Consequently, there is an invariant to respect as we write the instruction.

```
(define (STORE-ARGUMENT m m* rank)
  (lambda ()
    (m)
    (stack-push *val*)
    (m*)
    (let ((v (stack-pop)))
      (set-activation-frame-argument! *val* rank v) ) ) )
```

The case of **REGULAR-CALL** is clear except that we must simultaneously keep the function to invoke during the computation of its arguments and the current environment during the invocation itself. We had this:

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (let ((f *val*))
      (m*)
      (let ((sr *env*))
        (invoke f *val*)
        (set! *env* sr) ) ) ) )
```

After we invent a new register—***fun***—we can transform that definition into this one:

```
(define (REGULAR-CALL m m*)
  (lambda ()
    (m)
    (stack-push *val*)
    (m*)
    (set! *fun* (stack-pop))
    (stack-push *env*)
    (invoke *fun*)
    (set! *env* (stack-pop)) ) )
```

In passing, we notice that we have also redefined the calling protocol for functions. It no longer takes the activation record as an argument since that record is already in the register ***val***. You can see this in the current version of **FIX-CLOSURE**:

```
(define (FIX-CLOSURE m+ arity)
  (let ((arity+1 (+ arity 1)))
    (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*)) ) ) )
```

By adding registers, we can also linearize calls to primitives as well. We'll introduce the registers `*arg1*` and `*arg2*`. One of them is not necessarily different from `*fun*`, which is never used at the same time anyway. Consequently, we'll write `CALL3` like this:

```
(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 Customizing Instructions

Currently the twenty-five instructions that we're redefining generate thunks where the body of a thunk is a sequence of register effects. To get the instructions we want, we need to transform these twenty-five instructions so that they generate sequences of thunks that have only one unique effect: to modify one register, to push one value onto the stack, etc. By inverting our point of view in this way, we'll introduce the idea of a program counter, that is, a specialized register designating the next instruction to execute. If we have a program counter, we will also be able to define the function calling protocol more precisely, and thus eventually we'll be able to describe the mysteries of implementing `call/cc`, too.

Linearizing Assignments

Let's take the case of `SHALLOW-ARGUMENT-SET!`. That instruction was defined like this:

```
(define (SHALLOW-ARGUMENT-SET! j m)
  (lambda ()
    (m)
    (set-activation-frame-argument! *env* j *val*) ) )
```

To transform it into a sequence of instructions, we'll rewrite it as the following two functions:

```
(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*))) )
```

That first one, with the same name as before, composes various effects and returns the list of final instructions. The second function `SET-SHALLOW-ARGUMENT!` is specialized to write in an activation record.

Linearizing Invocations

REGULAR-CALL provides a good example of linearization. To customize all its components, we add the following instructions to the final machine: **PUSH-VALUE**, **POP-FUNCTION**, **PRESERVE-ENV**, **FUNCTION-INVOKE**, and **RESTORE-ENV**.

Here are those new definitions:

```
(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)))) )
```

Just as we wanted, now the result of compiling is a list of elementary instructions. However, this list is not directly executable, so we must provide an engine to evaluate this list of instructions. For that purpose, we define **run** like this:

```
(define (run)
  (let ((instruction (car *pc*)))
    (set! *pc* (cdr *pc*))
    (instruction)
    (run) ) )
```

Compiling now results in a list of instructions stored in the variable ***pc*** which plays the role of the program counter. The function **run** simulates a processor that reads an instruction, increments the program counter, executes the instruction, and then begins all over again. Incidentally, all the instructions here are represented by closures with the same signature `(lambda () ... )`.

Linearizing the Conditional

Linearizing a conditional is always problematic because of the two possible exits. How should we linearize a fork like that? To handle that, we'll (re)invent two new jump instructions that affect the program counter: **JUMP-FALSE** and **GOTO**. **GOTO**, familiar from other languages, represents an unconditional jump. **JUMP-FALSE** tests the contents of the register ***val*** and jumps only if it contains False. Both these instructions affect only the register ***pc*** to the exclusion of any other.

```
(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)))) )
```

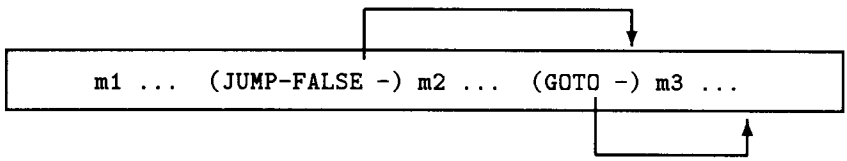


Figure 7.1 Linearizing the conditional

With those two new instructions, here's how we linearize the conditional. You can see it better in Figure 7.1.

```
(define (ALTERNATIVE m1 m2 m3)
  (append m1 (JUMP-FALSE (+ 1 (length m2))) m2 (GOTO (length m3)) m3) )
```

The condition is calculated and then tested by **JUMP-FALSE**. If the condition is true, we execute the instructions that follow it, and at the end of those computations, we jump over the corresponding instructions in the alternate. If the condition is false, we jump to the alternate that will be executed. Here you can see that we've just arrived at the level of an assembly language. Notice, however, that these jumps are relative to our current position, that is, they are program-counter relative.

Linearizing Abstractions

The last instruction that's hard to linearize is the one to create a closure. It's difficult because we have to splice together the code for the function with the code which creates it. Again, we'll use a jump to do this, as in Figure 7.2. Here's how we create a closure with variable arity:

```
(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 ) )
(define (CREATE-CLOSURE offset)
  (list (lambda () (set! *val* (make-closure (list-tail *pc* offset)
                                             *env* )))) )
(define (PACK-FRAME! arity)
  (list (lambda () (listify! *val* arity))) )
```

The new instruction **CREATE-CLOSURE** builds a closure for which the code is found right after the following **GOTO** instruction. Once the closure has been created and put into the register ***val***, its creator jumps over the code corresponding to its body in order to continue in sequence.

7.1.4 Calling Protocol for Functions

Functions are invoked by the instructions **TR-REGULAR-CALL** or **REGULAR-CALL**, which we covered a little earlier. [see p. 229] The function **invoke** condenses

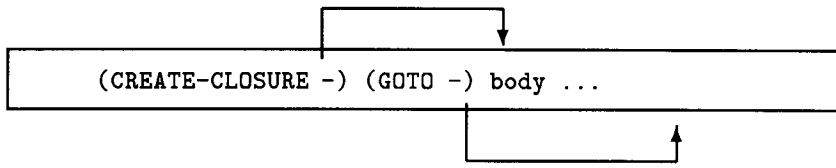


Figure 7.2 Linearizing an abstraction

that function calling protocol, like this:

```
(define (invoke f)
  (cond ((closure? f)
         (stack-push *pc*)
         (set! *env* (closure-closed-environment f))
         (set! *pc* (closure-code f)) )
        ... ) )
```

To invoke a function, we save the program counter that indicates the next instruction that follows the instruction **(FUNCTION-INVOKE)** in the caller. Then we take apart the closure to put its definition environment into the environment register ***env***. Eventually, we assign the address of the first instruction in its body to the program counter. We don't save the current environment because it has already been handled elsewhere according to whether the function was invoked by **TR-REGULAR-CALL** or **REGULAR-CALL**.

Then the function **run** takes over and executes the first instruction from the body of the invoked function to verify its arity, then, in case of successful verification, to extend the environment with its activation record. The activation record is in the register ***val***. By now, everything is in place for the function to evaluate its own body. A value is then computed and put into ***val***. Then that value has to be transmitted to the caller; that's the role of the **RETURN** instruction. It pops the program counter from the top of the stack and returns to the caller, like this:

```
(define (RETURN)
  (list (lambda () (set! *pc* (stack-pop)))) )
```

About Jumps

Assembly language programmers have probably noticed that we've been using only forward jumps. Moreover, the jumps as well as the construction of closures are all relative to the program counter. This relativity means that the code is independent of its actual place in memory. We call this phenomenon *pc-independent code* in the sense of independent of the program counter.

7.2 Language and Target Machine

Now we're going to define our target machine as well as the language for programming it. The machine will have five registers (***env***, ***val***, ***fun***, ***arg1***, and ***arg2***), a program counter (***pc***), and a stack, as you see in Figure 7.3.

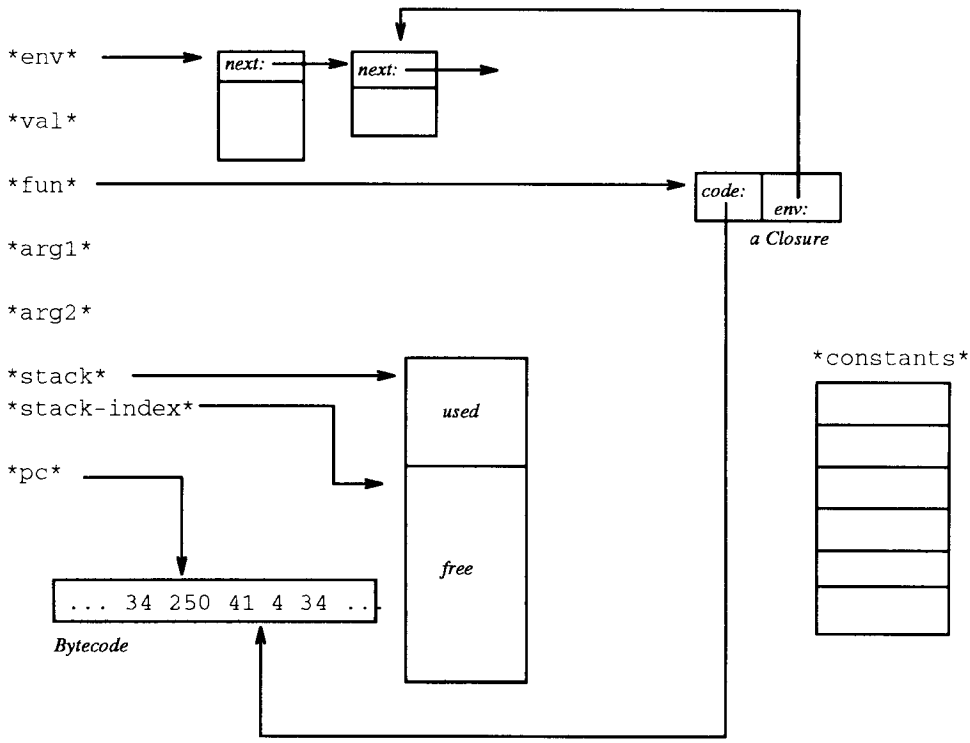


Figure 7.3 Byte machine

There are now thirty-four instructions. They appear in Table 7.2. In addition to the instructions you've already seen, we've added **FINISH** to complete calculations (or, more precisely, to get out of the function **run**) and to return control to the operating system or to its simulation in Lisp. [see p. 223]

The twenty-five instructions/generators of the intermediate language can be organized into two groups: leaf instructions and composite instructions. The nine leaf instructions are identified as such by the same name; they are marked by a star as a suffix in Table 7.2. In contrast, the sixteen composite instructions are defined explicitly in terms of the twenty-five new elementary instructions. We could have customized them more, for example, by decomposing **CHECKED-GLOBAL-REF** into a sequence of two instructions that carried out **GLOBAL-REF** and then verified that ***val*** actually contains an initialized value, but that would have slowed the interpreter. Conversely, we could have grouped some instructions together, as for **CALL3**: **(POP-ARG2)** is always followed by **(POP-ARG1)** so that particular sequence could be combined into one instruction.

(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 *offset*)
(GOTO *offset*)	(EXTEND-ENV)
(UNLINK-ENV)	(CALL0 *address*)*
(INVOKE1 *address*)	(PUSH-VALUE)
(POP-ARG1)	(INVOKE2 *address*)
(POP-ARG2)	(INVOKE3 *address*)
(CREATE-CLOSURE *offset*)	(ARITY=? *arity* + 1)
(RETURN)	(PACK-FRAME! *arity*)
(ARITY>=? *arity* + 1)	(POP-FUNCTION)
(FUNCTION-INVOKE)	(PRESERVE-ENV)
(RESTORE-ENV)	(POP-FRAME! *rank*)
(POP-CONS-FRAME! *arity*)	(ALLOCATE-FRAME *size*)*
(ALLOCATE-DOTTED-FRAME *arity*)*	(FINISH)

Table 7.2 Symbolic instructions

A few of the sixteen composite instructions of the intermediate language haven't appeared before, so here are their unadorned definitions with no explanation. Later we'll get to the definition of the missing machine instructions that appeared in Table 7.2.

```
(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* rank)
  (append m (PUSH-VALUE) m* (POP-FRAME! rank)) )
(define (CONS-ARGUMENT m m* arity)
  (append m (PUSH-VALUE) m* (POP-CONS-FRAME! arity)) )
```

The many **append**s that break up these definitions make pretreatment particularly costly and inefficient. A good solution is to build the final code in one pass. That's what we would do to compile into C. [see p. 379] To do that, we must linearize the code production; that's an activity that is independent of linearizing the code as we have done it in this chapter. To take just one example: for `CALL2`, we would do the following:

1. generate the code for `m1`;
2. produce the code for `(PUSH-VALUE)`;
3. generate the code for `m2`;
4. produce the code for `(POP-ARG1)`;
5. produce the code for `(INVOKE2 address)`.

Carrying out those modifications is trivial everywhere except in the `GOTO`s and `JUMP-FALSE`s. We can no longer precompute them in `ALTERNATIVE`, `FIX-CLOSURE`, nor `NARY-CLOSURE`. To handle that chore, we must implement *backpatching*. For example, in `FIX-CLOSURE`, we must do the following:

1. produce the instruction `CREATE-CLOSURE`;
2. produce a `GOTO` instruction without specifying the offset but noting the current value of the program counter;
3. generate the code for the body of the function;
4. note the current value of the program counter to deduce the offset of the `GOTO` we generated earlier;
5. write that offset in the place[1] reserved for it in the `GOTO`.

1. Things get a little complicated on a byte machine, depending on whether we reserve one or

The result will be a cleaner, more efficient process for producing code than the one we've come up with, but we kept ours around for its simplicity.

7.3 Disassembly

At the beginning of this chapter, [see p. 223] we showed the intermediate form of the following little program:

```
((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)))) )) )
```

Now we can show the symbolic form that it elaborates in our machine language. The 78 instructions of Figure 7.4 are really beginning to look like machine language.

7.4 Coding Instructions

For pages and pages, we've talking about bytes, but we've not yet actually seen one. To bring them out of hiding, we simply have to look at the instructions in Table 7.2 differently: instead of seeing them as instructions, we should regard them as byte generators executed by a new, more highly adapted **run**. This change in our point of view will highlight important improvements in the generated code, namely its speed and compactness.

Coding instructions as bytes has to be done very carefully. We must simultaneously choose a byte and associate it with its behavior inside the **run** function as well as with various other information, such as the length of the instruction (for example, for a disassembly function). For all those reasons, instructions will be defined by means of special syntax: **define-instruction**. We'll assume that all the definitions of instructions appearing here and there in the text have actually been organized inside the macro **define-instruction-set** along with a few utilities, like this:

```
(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)) ... ) ) )
       (define (instruction-decode code pc)
```

two bytes for the offset, since it might be more or less than 256, once it's determined. There is some risk here of suboptimality.

```
(CREATE-CLOSURE 2)
(GOTO 59)
(ARITY=?  4)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(PUSH-VALUE)
(CONSTANT 0)
(POP-ARG1)
(INVOKE2 =)
(JUMP-FALSE 10)
(SHALLOW-ARGUMENT-REF 2)
(PUSH-VALUE)
(CONSTANT 1)
(PUSH-VALUE)
(ALLOCATE-FRAME 2)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(GOTO 39)
(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 19)
(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 2)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 4)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 2)
(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 4)
(ARITY=?  2)
(EXTEND-ENV)
(SHALLOW-ARGUMENT-REF 0)
(RETURN)
(PUSH-VALUE)
(ALLOCATE-FRAME 4)
(POP-FRAME! 2)
(POP-FRAME! 1)
(POP-FRAME! 0)
(POP-FUNCTION)
(FUNCTION-GOTO)
(RETURN)
```

Figure 7.4 Compilation

```
        (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) ) )
```

With **define-instruction**, we can generate three functions at once: the function **run** to interpret bytes; the function **instruction-size** to compute the size of an instruction; the function **instruction-decode** to disassemble bytes composing a given instruction. **instruction-decode** is particularly useful during debugging. Let's consider one of those functions:

```
(define-instruction (SHALLOW-ARGUMENT-REF j) 5
  (set! *val* (activation-frame-argument *env* j)) )
```

That definition participates in the definition of **run** by adding a clause triggered by byte 5, like this:

```
(define (run)
  (let ((instruction (fetch-byte)))
    (case instruction
      ...
      ((5) (let ((j (fetch-byte)))
             (set! *val* (activation-frame-argument *env* j)) ))
      ... ) )
  (run) )
```

The function **fetch-byte** reads the necessary argument or arguments. Simply defined, it has a secondary effect of incrementing the program counter, like this:

```
(define (fetch-byte)
```

```
(let ((byte (vector-ref *code* *pc*)))
  (set! *pc* (+ *pc* 1))
  byte ) )
```

The program counter itself is handled by the function **run**, which also uses **fetch-byte** to read the next instruction. By the way, that's also the case with certain other processors: during the execution of one instruction, the program counter indicates the *next* instruction to execute, not the one currently being executed.

If we want to increase the speed at which bytes are interpreted (in other words, if we want a fast **run**), we have to keep an eye on the execution speed of the form **case** appearing there. Since an instruction can be only one byte, that is, a number between 0 and 255, the best compilation of **case** uses a jump table indexed by bytes because in that way choosing a clause to carry out takes constant time. If instead we expanded **case** into a set of (**if** (**eq?** ...) ...), then we would be obliged to use a linear search, a tactic that would be lethal in terms of execution speed. Few compilers for Lisp (but among them are Sqil [Sén91] and Bigloo [Ser93]) are capable of the performance we've designed here.

define-instruction also participates in the function **instruction-size**; it adds the clause so that an instruction **SHALLOW-ARGUMENT-REF** has two-byte length. The instruction is indicated by its address (**pc**) in the byte-code vector where it appears. The **case** form here is equivalent to the vector of instruction sizes.

```
(define (instruction-size code pc)
  (let ((instruction (vector-ref code pc)))
    (case instruction
      ...
      ((5) 2)
      ... ) ) )
```

define-instruction also participates in the function **instruction-decode** by adding to it a specialized clause to recognize **SHALLOW-ARGUMENT-REF**. The function **instruction-decode** uses its own definition of **fetch-byte** so that it does not disturb the program counter but it still resembles **run**, like this:

```
(define (instruction-decode code pc)
  (define (fetch-byte)
    (let ((byte (vector-ref code pc)))
      (set! pc (+ pc 1))
      byte ) )
  (let ((instruction (fetch-byte)))
    (case instruction
      ...
      ((5) (let ((j (fetch-byte))) (list 'SHALLOW-ARGUMENT-REF j)))
      ... ) ) )
```

7.5 Instructions

There are 256 possibilities in our instruction set, leaving us plenty of room since we need only 34 instructions. We'll take advantage of this bounty to set aside a few bytes for coding the most useful combinations. For example, it's already clear

that most of the functions have few variables, as [Cha80] observed. At the time I wrote these lines, I analyzed the arity of functions appearing in the programs associated with this book, and got the following results. Only 16 functions have variable arity; the 1,988 others are distributed as you see in Table 7.3.

arity	0	1	2	3	4	5	6	7	8
frequence (in %)	35	30	18	9	4	1	0	0	0
accumulation (in %)	35	66	84	93	97	99	99	99	100

Table 7.3 Distribution of functions by arity

A look at this table indicates that most functions have fewer than four variables. If we generalize these results, we have to admit that zero arity is over-represented here because of Chapter 6. With these observations in mind, we'll start looking for ways to improve the execution speed of functions having fewer than four variables. In consequence, all instructions involving arity will be specialized for arities less than four.

7.5.1 Local Variables

Among all the possibilities, let's first take the case of **SHALLOW-ARGUMENT-REF**. This instruction needs an argument, j, and it loads the register ***val*** with the argument j of the first activation record contained in the environment register ***env***. We can specialize it by dedicating five bytes to represent the cases $j = 0, 1, 2, 3$. We can also restrict our machine so that it does not accept functions of more than 256^2 variables. That limit will let us code the index of the argument to search for in only one byte. The function **check-byte**[3] verifies that. Here then is the function **SHALLOW-ARGUMENT-REF** as a generator of byte-code. It returns the list[4] of generated bytes. That list contains one or two bytes, depending on the case.

```
(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 (and (<= 0 j) (<= j 255))
    (static-wrong "Cannot pack this number within a byte" j) ) )
```

2. COMMON LISP has the constant `lambda-parameters-limit`; its value is the maximal number of variables that a function can have; this number cannot be less than 50. Scheme says nothing about this issue, and that silence can be interpreted in various ways. How to represent an integer is an interesting problem anyway. Most systems with *bignums* limit them to integers less than $(256)^{2^{32}}$, rather short of infinity. One solution is to prefix the representation of such a number by the length of its representation. This eminently recursive strategy stops short, of course, at the representation of a small integer.

3. We won't mention `check-byte` again in the explanations that follow, simply to shorten the presentation.

4. Once again, we're using up our capital by a profusion of `lists` and `appends`.

Here are the five[5] associated physical instructions:

```
(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! operates on local variables. Modifying local variables involves the same treatment and leads to what follows. (You can easily deduce the missing definitions.)

```
(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*) )
```

As for deep variables, we'll assume that all cases[6] are equally probable, so we'll code them like this:

```
(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*) )
```

7.5.2 Global Variables

We'll assume that all mutable global variables are equally probable and they can thus be coded directly. To simplify, we'll also assume that we can't have more than 256 such variables so that we can code each one in a unique byte. Thus we'll have this:

```
(define (GLOBAL-REF i) (list 7 i))
(define (CHECKED-GLOBAL-REF i) (list 8 i))
(define (SET-GLOBAL! i) (list 27 i))
```

5. There's no instruction for the code 0 because there are already too many zeroes at large in the world.

6. Well, in fact, that is a false assumption since the explanations we just offered at least show that the parameter j is generally less than four. However, deep variables are rare, and that fact justifies our not trying to improve access to them.

```
(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*) )
```

The case of predefined variables that cannot be modified is more interesting because we can dedicate a few bytes to the statistically most significant cases. Here, we'll deliberately accelerate the evaluation of the variables **T**, **F**, **NIL**, **CONS**, **CAR**, and a few others, like this:

```
(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)) )
```

Since we've begun by special treatment for a few constants, we'll treat quoting in the same way. Let's assume that the machine has a register, ***constants***, containing a vector that itself contains all the quotations in the program. Then the function **quotation-fetch** can search for a quotation there.

```
(define (quotation-fetch i)
  (vector-ref *constants* i) )
```

Quotations are collected inside the variable ***quotations*** by the combinator **CONSTANT** during the compilation phase. Those quotations will be saved during compilation and put into the register ***constants*** at execution. Once more, all quoted values are not equal; some are quoted more often than others, so we'll dedicate a few bytes to those. We'll reuse a few of the bytes we've already predefined, namely, **PREDEFINED0** and those that follow it. Finally, we'll assume that we can quote as immediate integers only those between 0 and 255. Other integers will be quoted[7] like normal constants.

```
(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 4)  (list 84))
```

7. That's annoying, but it still lets us implement *bignums* as lists.

```
        ((and (integer? value)  ;immediate value
              (<= 0 value)
              (< value 255) )
         (list 79 value) )
        (else (EXPLICIT-CONSTANT value)) ) )
(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) )
```

7.5.3 Jumps

If you think the restrictions we've imposed so far are too limiting, wait until you see what we do about jumps. We must not limit `GOTO` and `JUMP-FALSE` to 256-byte jumps[8] since the size of a jump depends on the size of the compiled code. We'll distinguish two cases: whether the jump takes one byte or two. We'll thus have two[9] physically different instructions: `SHORT-GOTO` and `LONG-GOTO`. This will, of course, be a handicap for our compiler that it won't know how to leap further than 65,535 bytes.

```
(define (GOTO offset)
  (cond ((< offset 255) (list 30 offset))
        ((< offset (+ 255 (* 255 256)))
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
           (list 28 offset1 offset2) ) )
        (else (static-wrong "too long jump" offset)) ) )
(define (JUMP-FALSE offset)
  (cond ((< offset 255) (list 31 offset))
        ((< offset (+ 255 (* 255 256)))
         (let ((offset1 (modulo offset 256))
               (offset2 (quotient offset 256)) )
           (list 29 offset1 offset2) ) )
        (else (static-wrong "too long jump" offset)) ) )

(define-instruction (SHORT-GOTO offset) 30
  (set! *pc* (+ *pc* offset)) )
(define-instruction (SHORT-JUMP-FALSE offset) 31
  (if (not *val*) (set! *pc* (+ *pc* offset))) )
(define-instruction (LONG-GOTO offset1 offset2) 28
```

8. Old machines like those based on the 8086 had that kind of limitation.
9. The same technique of doubling an instruction in `SHORT-` and `LONG-` could be applied to `GLOBAL-REF` and its companions so that the number of mutable global variables would no longer be limited to 256.

```
(let ((offset (+ offset1 (* 256 offset2))) )
  (set! *pc* (+ *pc* offset)) ) )
```

7.5.4 Invocations

We need to analyze first general invocations and then inline calls. The reason we favored functions with low arity is still valid here. For a few bytes more, we can specialize the allocation of activation records for arities up to four variables.

```
(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)) )
```

How we put the values of arguments into activation records can also be improved for low arities, like this:

```
(define (POP-FRAME! rank)
  (case rank
    ((0 1 2 3) (list (+ 60 rank)))
    (else      (list 64 rank)) ) )

(define-instruction (POP-FRAME!0) 60
  (set-activation-frame-argument! *val* 0 (stack-pop)) )
(define-instruction (POP-FRAME! rank) 64
  (set-activation-frame-argument! *val* rank (stack-pop)) )
```

Inline calls are very frequent, so they warrant a few dedicated bytes themselves. For example, **INVOKE1**, the special invoker for predefined unary functions, is written like this:

```
(define (INVOKE1 address)
  (case address
    ((car)     (list 90))
    ((cdr)     (list 91))
    ((pair?)   (list 92))
    ((symbol?) (list 93))
    ((display) (list 94))
    (else (static-wrong "Cannot integrate" address)) ) )

(define-instruction (CALL1-car) 90
  (set! *val* (car *val*)) )
(define-instruction (CALL1-cdr) 91
  (set! *val* (cdr *val*)) )
```

Of course, we'll do the same thing for predefined functions of arity 0, 2, and 3. The reason that **display** appears in **INVOKE1** is connected with debugging; it's actually a function that belongs in a library of non-primitives because of its size and the amount of time it requires when it's applied. It would be better to

dedicate a few bytes to **cddr**, then to **cdddr**, and finally to **cadr** (in the order of their usefulness).

Verifying arity itself can be specialized for low arity, so we'll distinguish these cases:

```
(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")) ) )
```

Taking into account the low proportion of functions with variable arity, we'll reserve the previous treatments for functions of fixed arity. So far, the inline functions we've distinguished are very simple and their computation is quick. Since we still have some free bytes, we could inline more complicated functions, such as, for example, **memq** or **equal**. The risk in doing so is that the invocation of **memq** might never terminate (or just take a really long time) if the list to which it applies is cyclic. MacScheme [85M85] limits the length of lists that **memq** can search to a few thousand.

7.5.5 Miscellaneous

There's still a number of minor instructions that we will simply code as one byte because they have no arguments. Byte generators are all cut from the same cloth, so to speak, so here's one example:

```
(define (RESTORE-ENV) (list 38))
```

And here are their definitions in terms of instructions:

```
(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) )
```

The following functions define how to save and restore the environment:

```
(define (preserve-environment)
  (stack-push *env*) )
(define (restore-environment)
  (set! *env* (stack-pop)) )
```

There are two methods to invoke functions: **FUNCTION-GOTO** in tail position and **FUNCTION-INVOKE** for everything else. For tail position, the stack holds the return address on top. For a non-tail position and after a function has been called, the machine must get back to the instruction that follows **(FUNCTION-INVOKE)** to carry on with its work. For that reason, **FUNCTION-INVOKE** must save the return program counter. When the call is in tail position, the instruction **FUNCTION-GOTO** should have already been compiled into **FUNCTION-INVOKE** followed by **RETURN**, but there's no point in pushing the address of **RETURN** onto the stack; it's sufficient not to put it there in the first place. Accordingly, **FUNCTION-GOTO** invokes a function without saving the caller since the caller has finished its work and has nothing left to do. Here's the generic definition of **invoke**. We hope it clarifies this explanation.

```
(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)) )
```

The function **invoke** is generic so that it can be extended to new types of objects and, for example, to primitives that we represent as thunks.

```
(define-method (invoke (f primitive) tail?)
  (unless tail? (stack-push *pc*))
  ((primitive-address f)) )
```

There again, the value of the variable **tail?** determines whether or not we save the return address.

7.5.6 Starting the Compiler-Interpreter

To produce an interpretation loop (as we've done in all the preceding chapters) we need finely tuned cooperation between a compiler and an execution machine. The function **stand-alone-producer7d** takes a program and initializes the compiler. By "initializes the compiler," we mean in particular its environment of predefined mutable global variables and the list of quotations. The result of the function

meaning is now a list of bytes that we put into a vector preceded by the code for **FINISH** and followed by the code corresponding to **RETURN**. The initial program counter is computed to indicate the first byte that follows the prologue. (Here, that's the second byte, that is, the one with the address 1.) For reasons that you'll see later [**see Ex. 6.1**] but that have already been explained, we'll prepare the list of mutable global variables and the list of quotations. Eventually, the final product of the compilation is represented by a function waiting to evaluate what we provide as the size of the stack.

```
(define (chapter7d-interpreter)
  (define (toplevel)
    (display ((stand-alone-producer7d (read)) 100))
    (toplevel) )
  (toplevel) )
(define (stand-alone-producer7d 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) )
```

The function **run-machine** initializes the machine and then starts it. It allocates a vector to store values of mutable global variables; it organizes the names of these same variables; it allocates a working stack, and it initializes all the registers. Now our only problem is stopping the machine! The program is compiled as if it were in tail position, that is, it will end by executing a **RETURN**. To get back, the stack must initially contain an address to which **RETURN** will jump. For that reason, the stack initially contains the address of the instruction **FINISH**. It's defined like this:

```
(define-instruction (FINISH) 20
  (*exit* *val*) )
```

When that instruction is executed, it stops the machine and returns the contents of its ***val*** register by means of a judicious escape set by **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)
```