# An Introduction to Scheme and its Implementation

Paul R. Wilson, University of Texas
wilson@cs.utexas.edu
http://www.cs.utexas.edu/users/wilson

# 1 Overview

This book provides an introduction to Scheme for programmers—it is not for first-time programmers, but for people who already know how to program (at least a little) and are interested in learning Scheme.

## 1.1 Scheme: A Small But Powerful Language

[ need to improve this introductory blather...]

Scheme is a clean and fairly small but powerful language, suitable for use as a general-purpose programming language, a scripting language, an extension language embedded within applications, or just about anything else.

Scheme was designed to lend itself to a variety of implementation strategies, and many implementations exist—most of them free software. There are straightforward interpreters (like BASIC or Tcl), compilers to fast machine code (like C or Pascal), and compilers to portable interpretive virtual machine code (like Java).

Several extended implementations of Scheme exist, including our own RScheme system, an extremely portable implementation of Scheme with an integrated object system and powerful extensibility features.

This is the first of three planned documents on Scheme, Scheme implementation, and the RScheme language and its implementation. When they're all finished, I may combine them into a big book. All three will be in Texinfo format, so that they can be printed out as hardcopy manuals, browsed online as info documents (with the Info browser, or the Info system for the Emacs editor), or converted automatically to HTML format for browsing with a web browser. Whichever way you're reading, this, welcome to Scheme.

*[ note: the current draft is only available in postScript form, because I haven't done all of the hyperlinking for the Info and HTML versions. ]*

## 1.2 Who Is this Book For?

This book is for people who are interested in how Scheme works, or people who are interested in Scheme in terms of programming language design—as well as people who are just interested in using Scheme.

There's not much conflict between these goals, since one of the best ways to learn Scheme—and important principles of language design—is to see how to implement Scheme, in Scheme. I'll illustrate the power of Scheme by showing a couple of simple interpreters for subsets of Scheme, and a simple compiler. A compiler for Scheme can be surprisingly simple and understandable.

This is a fairly traditional approach, pioneered by Abelson and Sussman in *Structure and Interpretation of Computer Programs*, which is a widely used and excellent introductory programming text. This approach has been followed, more or less, in several other introductory books on Scheme programming. Most of those books, though, are for beginning programmers. While I think Scheme is a great first language, there are many people out there who've had to suffer through C or Pascal or whatever, and don't want to wade through an introductory programming book just to learn Scheme.

My approach is different from most of the current books on Scheme, in several ways. *[When it's finished, this book will be hypertext, and can be kept online in online for handy reference in any of several cross-indexed formats...]*

I will breeze through basic programming ideas—for example, I assume you have some idea what a variable is, and what recursion is.

I take a more concrete approach than many Scheme writers do, because I've found many students find it easier to understand. Every now and then I'll dip below the language level, and tell you how most actual implementations of the language work. I find that this concreteness helps disambiguate things in many students' minds—as well as in my own.

I do not start from a functional programming perspective that pretends that Scheme executes by rewriting expressions. (If that doesn't mean anything to you, definitely don't worry about it!)

I take Scheme to be a special case of a *weakly object-oriented* procedural language. By weakly object oriented, I don't mean that it's object-oriented in the sense of having inheritance and so on—though several extended versions of Scheme do. I just mean that the values in the language are data objects (records, etc.) whose *identities* may be significant—that is, you can compare pointers to two objects to see whether they are the *very same* object, not just and whether they have the same state—and objects may have mutable (changable) state. (This view is developed further in RScheme, which is a fully object-oriented language that happens also to be Scheme. But that's a different book, not yet written.)

Some people may not like this approach, since I start talking about state and assignment very early. It is generally considered bad style in Scheme to use assignments freely, and good style to write mostly "functional" or "applicative" programs. While I agree that mostly-functional programming is usually the right thing for Scheme, my intent is to make the semantics of the language clear early on, and to make it clear to new Schemers that Scheme is a fairly normal programming language, even if it is unusually clean and expressive. My experience in teaching Scheme has convinced me that many people benefit from an early exposure to the use of assignment; it clarifies fundamental issues about variables and variable binding. Style is discussed later, when alternatives are clearer.

If you've ever tried to learn Lisp or Scheme before, but not gotten very far, this book may be for you. Many people take to Lisp or Scheme like ducks to water. Some people don't, however, and I think that's often because of the way that the material is presented—there's nothing hard about learning Lisp or Scheme. In this book, I try to explain things a little differently than they're usually explained, to avoid the problems that some people have learning from most of the existing books. The concreteness of the explanations here may help overcome the unfamiliarity of these languages. Scheme is really just a normal programming language, but one with powerful features that can be used in special ways, too.

If you're a programming language designer, but not fluent in Scheme or Lisp, this book may help clarify what these languages are all about. It's my belief that there has been a damaging split between the Lisp world and the "conventional" imperative programming language world, largely due to the different vocabularies of the different communities. Recent developments in Scheme have not been widely appreciated by designers of other languages. (This theme will be developed further in the other documents in this series.) Even old features of Lisp, such as macros, have not been properly understood by language designers in general, and their problems have been substantially solved in Scheme.

If you're a programming language implementor, or teaching programming language implementation, this book may be of use. (I use it in a course on languages and implementation.) I'll present interpreters and a compiler for Scheme. Scheme an excellent vehicle for teaching principles of language implementation, because its syntax is simple, and there is a straightforward evolution from simple interpreters to more complex ones, and another straightforward move from a simple interpreter to a compiler. This supports teaching the principles of language implementation with a minimum of irrelevant detail.

## 1.3 Why Scheme?

[ Warn people that this is partisan propaganda... ]

Scheme is a very nice language for implementing languages, or for transformational programming in general—that is, writing programs that write programs—or for writing programs that can easily be extended or customized. The features that make Scheme attractive for implementing Scheme also make it good for all kinds of things, including scripting, the construction of new languages and application-specific programming environments, and so on.

[ As you learn Scheme, you'll probably realize that all interesting programs end up being, in effect, application-specific programming environments...]

Most Scheme systems are interactive, allowing you to incrementally develop and test parts of your program. In this respect, it is much like BASIC or Tcl—but a far cleaner and more expressive language. Scheme can also be compiled, to make programs run fast. This makes it easy to develop in, like BASIC or Tcl, but still fast, like C. (Scheme isn't usually quite as fast as C, but it's usually not too much slower, if you get a good Scheme compiler.) So if you're a Tcl or BASIC programmer looking for a less crufty and/or fossilized language, Scheme may be for you.

Unlike most interactive languages, Scheme is well-designed: it's not a kludge cobbled up by some people with very limited applications in mind, and later extended past its reasonable scope of application. It was designed from the outset as a general-purpose language, combining the best features of two earlier languages. It is fairly radical revision of Lisp, incorporating the best features of both Lisp and Algol (the ancestor of C, Pascal, et al.).

(This is why Scheme has been adopted by several groups as an alternative to kludgey languages like Tcl and Perl. The Free Software Foundation's Guile extension language is based on Scheme. So is the Scheme Shell (`scsh`), which is a scripting language for UNIX. The CAD Framework Initiative has adopted Scheme as the glue for controlling Computer-Aided Design tools. The Dylan language is also based on Scheme, though with a different syntax and many extensions.)

If you want to learn Lisp, Scheme is a good place to start. Common Lisp is a big, somewhat messy language, which is probably easiest to learn by starting with Scheme. Then you can understand Common Lisp as a series of extensions (and significant obfuscations) of Scheme. Some of the best features of Common Lisp were copied from Scheme.

If you want to get something of the flavor of functional programming, you can do that in Scheme—most well-written Scheme programs are largely functional, because that's simply the easiest way to do many interesting things.

And if you just want to learn to program better, Scheme may open your eyes to new ways of thinking about programs. Many people prototype programs in Scheme, because it's so easy, even if they eventually have to recode them in other languages to satisfy their employers.

## 1.4 Why Scheme Now?

Scheme is not a new language—it's been around and evolving slowly for 20 years.

The evolution of Scheme has been slow, because the people who standardize Scheme have been very conservative—features are only standardized when there is a near-universal consensus on how they should work. The focus has been on quality, not industrial usability.

This policy has had two consequences. The first is that Scheme is a beautiful, extremely well-designed language. The second is that Scheme has been "behind the curve," lacking several features that are useful in general-purpose languages. Gradually, though, Scheme has grown from a very small language, suitable only for teaching concepts, to a very useful language.

The most important new feature of Scheme (in my view) is lexically-scoped ("hygeinic") macros, which allow the implementation of many language features in a portable and fairly efficient way. This allows Scheme to remain small, but also allows useful extensions to the base language to be written as libraries, without a significant performance penalty.

## 1.5 What this Book Is Not

This book isn't a language definition for Scheme, or a manual for using any particular Scheme implementation. There is a free language definition document for Scheme, easily available via the internet, called the Revised Scheme Report. (There's also an IEEE standard.) I recommend getting the Scheme report and printing it out, or browsing the html version with a web browser. (`http://www.cs.indiana.edu/scheme-repository/doc.standards.html` It's not very big, because Scheme is a pretty small language. I also recommend having a look at the documentation for the particular implementation of Scheme you're using.

On the other hand, this book may serve as a passable approximation of a language manual most of the time. (It may work better for this purpose once it's fleshed out more and I've devised more online indexing.) It describes all of the important features of standard Scheme, clearly enough that you can use them for most purposes. This is possible because Scheme is very clean and "orthogonal"—most of its features don't interact in surprising ways, so if you understand Scheme, and do the "Scheme-ish" thing, Scheme will generally do what you expect.

For more information on Scheme, particular Scheme implementations, and so on, see the FAQ (Frequently Asked Questions) List on the usenet newsgroup `comp.lang.scheme`. It's available from the Scheme Repository via anonymous internet ftp from `ftp.cs.indiana.edu` in the directory `pub/scheme-repository`. Or if you're a World Wide Web user, visit the Scheme repository at `http://www.cs.indiana.edu/scheme-repository`. The Scheme repository contains several free implementations of Scheme, as well as a variety of useful programs, libraries, and papers.

## 1.6 Structure of this Book

This book's structure reflects its tutorial intent, rather than any strong grouping of concepts. In the next three chapters, ideas are introduced in the order that I think they're easiest to learn. Each chapter introduces a few more or less related ideas, with small code examples, and ends with more examples of Scheme programs to show why the ideas are useful. The later chapters introduce relatively independent topics.

[ The following needs to be reworked a little, after the actual document structure settles down. ]

Chapter 2 [Introduction], page 9 describes some basic features of Scheme, including a little syntax, and gives code examples to show that Scheme can be used like most programming languages—you don't give up much when using Scheme, and it's not hard to switch.

Chapter 3 [Using Scheme (A Tutorial)], page 83 gives a tutorial on Scheme programming, intended to be used while sitting at a running Scheme system and trying examples interactively.

Chapter 4 [Writing an Interpreter], page 149 presents an simple interpreter for a subset of Scheme.

Chapter 5 [Environments and Procedures], page 167 describes Scheme's binding environments and procedures, and shows how procedural abstraction can be very powerful in a language with first-class procedures, block structure indefinite extent (garbage collection). It then shows an implementation of binding environements and procedures for the interpreter from the previous chapter, and shows how to use Scheme's binding and procedure-defining constructs in fairly sophisticated ways.

Section 6.3.2 [Recursion in Scheme], page 218 discusses recursion, and especially *tail recursion*.

Chapter 7 [Quasiquotation and Macros], page 223 presents quasiquotation, a means of constructing complex data structures and variants of stereotyped data structures, and then presents macros, a facility for defining your own "special forms" in Scheme. Macros let you define your own control constructs, data-structuring systems such as object systems, etc. (If you've ever been daunted by problems with C or Lisp macros, don't worry—Scheme macros fix the major problems with older macro systems.) Macros are also interesting because they're often used in the implementation of Scheme itself. They allow the language implementation to be structured in a layers, with most of the language written in the language itself, by bootstrapping up from a very small core language understood by the compiler.

Chapter 9 [Other Useful Features], page 254 presents a variety of miscellaneous features of Scheme that are useful in writing real programs. They're not part of the conceptual core of Scheme, but any useful language should have them.

Section 8.2.2.2 [Records and Object Orientation], page 253 ...

Section 10.6 [call-with-current-continuation], page 262 discusses first-class continuations, the most powerful control construct in Scheme. Continuations allow you to capture the state of the activation stack (sort of), and return to that state to resume at a given point in a program's execution. Continuations are conceptually weird, and are not to be used casually, but tremendously expressive for things like backtracking, threads, etc.

Section 11.15.7 [Compiling Scheme], page 309 presents an example Scheme program that happens to be a simple compiler for Scheme. It's a "toy" compiler, but a real compiler nonetheless, with all of the basic features of any Scheme compiler, but minimal boring "support" hacks to perform tokenization, storage management, etc.

# 2 Introduction

In this chapter, I'll give a quick overview of some basic features of Scheme, enough to get started writing some programs.

This chapter moves fairly quickly, briefly introducing about half of the ideas in Scheme. In later chapters, I'll explain and demonstrate these features more fully, and introduce other advanced features.

This chapter is meant to be read concurrently with the first half of the next one, which includes a tutorial on using Scheme interactively. I've put in directives saying when you should work through parts of the next chapter. After becoming familiar with Scheme, it will serve as a basic reference; you can consult the next chapter for basic examples, and later chapters for advanced techniques.

If you're fluent in concepts of programming languages, and especially if you've programmed in Lisp, you may be able to breeze through this chapter to get a sense of what Scheme is about. If you're fluent in programming language concepts, you may be able to read straight through this section.

*(NOTE TO MY CS345 STUDENTS: don't try to breeze through this. Do the tutorial hunks after each hunk of this chapter.)*

If you intend to actually program in Scheme, you should definitely follow the directives and read parts of the next chapter, rather than trying to plow straight through this one.

## 2.1 What is Scheme? (Hunk A)

```
==================================================================
     Hunk A starts here:
==================================================================
```

First, a bunch of jargon—ignore it if you want to:

Scheme is a lexically-scoped, block structured, dynamically typed, mostly functional language. It is a variant of Lisp. It has first-class procedures with block structure and indefinite extent. Parameter passing is by value, but the values are references. It has first-class continuations to allow the construction of new control abstractions. It has lexically-scoped ("hygeinic") macros to allow definition of of new syntactic forms, or redefinition of old ones.

If none of that means anything to you right now, don't worry. Keep reading.

Scheme is designed to be an *interactive* and *safe* language. A normal Scheme system is really an interactive program that you can use to run *parts* of your Scheme program in the order you want. When one has run, your program doesn't just terminate, and your data don't disappear—Scheme askes you what to do next, and you can examine the data or tell Scheme to run another part of the program.

Scheme is *safe* in that the interactive system generally won't crash. If you make a mistake that would crash the system, Scheme detects that, and asks you what to do about it. It lets you examine and change the system's state, and go on. This is a very different style of programming and debugging from the normal edit-compile-link-run-crash cycle of "batch" programming languages like C and C++.

## 2.2 Basic Scheme Features

I'll go briefly through some of the basic features of Scheme, giving little code examples for clarity.

### 2.2.1 Code Consists of Expressions

Like Lisp, Scheme is written as prefix expressions, with parentheses for grouping. Prefix means that the name of an operation comes first, before its operands (the things it operates on).

In Scheme, there's no distinction between expressions (like arithmetic operations) and statements (like an `if` or a loop or a declaration). They're all "expressions"—it's a very general term.

#### 2.2.1.1 Parenthesized Prefix Expressions

In C or Pascal, a call to procedure `foo` with arguments `bar` and `baz` is written

```
foo(bar,baz);
```

but in Scheme it's written

```
(foo bar baz)
```

Note that the procedure name goes *inside* the parentheses, along with the arguments. Get used to it. It may seem less odd if you think of it as being like a operating system shell command—e.g., `rm foo`, or `dir bar`—but delimited by parentheses.

Just as in C, expressions can be nested. Here's a call to a procedure `foo`, with nested procedure call expressions to compute the arguments.

```
(foo (bar x) (baz y))
```

This is pretty much equivalent to C's

```
foo(bar(x),baz(y));
```

As in C or Pascal, the argument expressions in a procedure call are evaluated *before* actually calling the procedure; the resulting values are what's passed to the procedure. In Scheme terminology, we say that the procedure is *applied* to the actual argument values.

You'll notice soon that Scheme has very few special characters, and that expressions are generally delimited by parentheses or spaces. For example, `a-variable` is a single identifier, not a subtraction expression. Identifiers in Scheme can include not only alphabetic characters and digits, but several other characters, such as `!`, `?`, `-`, and `_`. Long identifiers are often constructed from phrases, to make it clear what they mean, using hyphens to separate words; for example, you can have a variable named `list-of-first-ten-lists`. You can use characters like `+`, `-`, `*`, and `/` within an identifier, as in `before-tax-total+tax`, or `estimate+epsilon`.

One consequence of Scheme's liberal rules for constructing identifiers is that *spaces are important*. You must put one or more spaces (or carriage returns) between identifiers except where special characters (usually parentheses) make the divisions obvious. For example, the addition expression `(+ 1 a)` can't be written `(+1 a)` or `(+1a)` or `(+ 1a)`. (It *can* be written `( + 1 a )`, because extra whitespace between tokens is ignored.)

#### 2.2.1.2 Expressions Return Values, But May Have Side-Effects

Scheme expressions combine the features of expressions and statements. They return values, but they can also have *side effects*—i.e., they can change the state of variables or objects by assignment.

The variable assignment operation in Scheme is `set!`, pronounced "set-bang." If we want to assign the value 3 to the variable `foo`, we write

```
(set! foo 3)
```

which is pretty much equivalent to C's

```
foo = 3;
```

Note that `(set! foo 3)` looks like a function call, because everything uses prefix notation, but it's not really a call; it's a different kind of expression.

You should not use assignments a lot in Scheme programs. It's usually a sign of bad style, as I'll explain later. I'll also show how to program in a style that doesn't need side effects much. They're there if you need them, though.

When you write a procedure that modifies its arguments, rather than just just returning a value, it's good style to give it a name that ends with an exclamation mark. This reminds you and anybody reading your code that the procedure changes something that already exists, rather than just returning a value such as a new data structure. Most of the standard Scheme procedures that change state are named this way.

Most Scheme procedures don't modify anything, however. For example, the standard procedure `reverse` takes a list as its argument and returns a list of the same elements in the opposite order. That is it returns a kind of reversed *copy* of the original list, without modifying the original at all. If you wrote a procedure that returned the *same* list, but modified so that its elements were in the opposite order, you'd probably call it `reverse!`. This warns people that a list that is passed to `reverse!` may be changed.

One side-effecting procedure we'll use in examples is `display`. `display` takes a value and writes a printed representation to the screen or a file. If you give it one argument, it writes to the "standard output"; by default, that's the terminal or other display.

For example, if you want to show the user the printed representation of the number 1022, you can use the expression

```
(display 1022)
```

The side effect of executing this expression is to write the `1022` on the user's screen. (`display` automatically converts the number to a string of characters so that you can read it.)

Note that `display` doesn't have an exclamation point at the end of its name, because it doesn't side-effect the argument you give it to print. You can give it a data structure and be sure that it won't modify it; display *does* have a side-effect, though—it changes the state of the screen (or file) that it writes to.

`display` is fairly flexible, and can write the printed representations of many common Scheme objects, and even fairly complex data structures.

Among many other things, display can print character strings. (Strings are another kind of Scheme object. You can write a literal string in double quotes, `"like this"`, and Scheme constructs a string object to hold that character sequence.

The expression `(display "Hello, world!)` has the side effect of writing `Hello, world!` to the standard output, which is usually the user's screen.

This makes `display` very useful for debugging, and for little examples, as well as for writing interactive programs. A similar procedure, `write` is used for saving data structures to files; they can then be copied back into memory using `read`.

In a later chapter, I'll show how to write to files by passing a second argument to `display` that tells it where to send the output. For now, you should just use `display` with *exactly one* argument. *Don't* try to pass `display` several things and expect it to print them all.

#### 2.2.1.3 Defining Variables and Procedures

You can define a variable in Scheme using a `define`:

```
(define my-variable 5)
```

This tells Scheme to allocate space for `my-variable`, and initialize that storage with the value `5`.

In Scheme, you always give a variable an initial value, so there's no such thing as an uninitialized variable or an unininitialized variable error.

Scheme values are always pointers to objects, so when we use the literal `5`, Scheme interprets that as meaning *a pointer to the object* `5`. Numbers are objects you can have pointers to, just like any other kind of data structure. (Actually, most Scheme implementations use a couple of tricks to avoid pointer overheads on numbers, but that doesn't show up at the language level. You don't have to be aware of it.)

After the above definition, we can draw the resulting situation like this:

```
     +-------+
 foo |   *---+--->5
     +-------+
```

The `define` expression does three things:

- It *declares* to Scheme that we're going to have a variable named `foo` in the current scope. (I'll talk about scoping a lot, later.)
- It tells Scheme to actually allocate storage for the variable. The storage is called a *binding*—we "bind" the variable `foo` to a particular piece of memory, so that we can refer to that storage by the name `foo`.
- It tells Scheme what initial value to put in the storage.

These three things happen when you define variables in other languages, too. In Scheme we have names for all three.

In the picture, the box represents the fact that Scheme has allocated storage for a variable. The name `foo` beside the box means that we've given that storage the name `foo`. The arrow says that the value in the box is a pointer to the integer object `5`. (Don't worry about how the integer object is actually represented. It doesn't really matter.)

You can define new procedures with `define`, too:

```
(define (two-times x)
   (+ x x))
```

Here we've defined a procedure named `two-times`, which takes one argument, `x`. It then calls the addition procedure `+` to add the argument value to itself, and returns the result of the addition.

Notice the sytnactic difference between the variable definition and the procedure definition: for a procedure definition, there are parentheses around the name, and the argument name(s) follow that inside the parentheses.

This resembles the way the procedure is called. Consider the procedure call expression `(two-times 5)`, which returns `10`; it looks like the definition's `(two-times x)`, except that we've put the *actual* argument 5 in place of the *formal parameter* `x`.

Here's a bit of programming language terminology you should know: the arguments you pass to a procedure are sometimes called *actual parameters*. The argument variables *inside* the procedure are called *formal parameters*—they stand for whatever is actually passed to the procedure at run time. "Actual" means what you actually pass to the procedure, and "formal" means what you call that on the inside of the procedure. Usually, I'll just talk about "arguments," but that's the same thing as "actual parameters." Sometimes I'll talk about "argument variables," and that's the same thing as "formal parameters."

You can define a procedure of zero arguments, but you still have to put parentheses around the procedure name, to make it clear that you're defining a procedure. You put parentheses around its name when you call it, too, to make it clear that it's a procedure call.

For example, this is a definition of a variable whose initial value is `15`:

```
(define foo 15)
```

but this is a definition of a procedure `foo` which returns `15` when called.

```
(define (foo) 15)
```

```
       +-------+
  foo  |   *---+--->#<procedure>
       +-------+
```

This picture shows that when you define a procedure, you're really defining a variable *whose value happens to be a (pointer to a) procedure*. For now, you don't really have to worry about that. The main thing to know is that now you can call the procedure by the name `foo`. For example, the procedure call expression `(foo 2)` will return `15`, because all the body of the procedure does is return the value `15`.

Usually, we indent procedure definitions like this, with the body starting a new line, and indented a few characters:

```
(define (foo)
   15)
```

This makes it clearer that it's a procedure definition.

#### 2.2.1.4 Most Operators are Procedures

In conventional programming languages like C and Pascal, there's an awkward distinction between procedure calls and other kinds of expressions. In C, for example, `(a + b)` is an expression, but `foo(a,b)` is a procedure call. In C, you can't do the same things with an operator like `+` that you can do with a procedure.

In Scheme, things are much more uniform, both semantically and syntactically. Most basic operations such as addition are procedures, and there is a unified syntax for writing expressions—parenthesized prefix notation. So rather than writing `(a + b)` in Scheme, you write `(+ a b)`. And rather than writing `foo(a,b)`, you write `(foo a b)`. Either way, it's just an operation followed by its operands, all inside parentheses.

For any procedure call expression (also called a *combination*), all of the values to be passed are computed before the actual call to the procedure. (This is no different from C or Pascal.)

### 2.2.2 Definitions vs. Assignments

Notice that we can give a variable a value in two ways: we can define it, specifying an initial value, or we can use `set!` to change its value.

The difference between these two is that `define` allocates storage for a variable, and gives that storage a name. `set!` does not. You must always `define` a variable before `set!` will work on it.

For example, if there's not already a definition of `quux`, the expression `(set! quux 15)` is an error, and Scheme will complain. You're asking Scheme to put (a pointer to) 15 in the storage named by `quux`—but `quux` doesn't name any storage yet, so it makes no sense.

It's rather like I'd told you, "give this to Philboyd" and handed you some object, (say, a pencil). If you don't know anybody named Philboyd, you're probably going to complain. `set!` is like that. We have to agree on what the word "Philboyd" means to before it makes sense to ask you to do something to Philboyd. `define` is a way of giving meaning to an identifier—making it refer to a piece of storage—as well as giving a value to put there.

#### 2.2.2.1 Special Forms

While most operations in Scheme are procedure calls, there are a few other kinds of expressions you need to know about, which behave differently. They are called *special forms*.

You've already seen two of the five or six important special forms, `define` and the assignment operator `set!`.

Notice that `set!` isn't a procedure, because its first argument is not really an expression to be evaluated in the normal way, to get a value to pass as an argument. It's the name of a place to *put* a value. (e.g., if we say `(set! a b)`, we get the value of `b`, and put it into the *storage named by* `a`.)

Likewise, `define` treats its first argument specially—the name of a variable or procedure isn't an expression that is evaluated and passed to `define`—it's just a name, and you're telling define to allocate some storage and use that name for it.

Other special forms we'll see include control constructs, like `if` and `do`, and forms for defining local variables, like `let`.

#### 2.2.2.2 Control Structures are Expressions

Scheme control structures are expressions, and return values. An `if` expression is a lot like a C if-then statement, but the "then" branch and the "else" branch are also expressions that return values; the `if` expression returns the value of whichever subexpression it evaluates.

For example,

```
(if (< a b)
    a
    b)
```

returns the value of either the variable `a`, or the variable `b`, whichever is less (or the value of `b` if they're equal). If you're familiar with ternary[1] expressions in C, this is like `(a < b) ? a : b`. In Scheme, there's no need for both an `if` statement and an if-like ternary expression operator, because `if` "statements" are expressions.

Note that even though every expression returns a value, not all values are used—you can ignore the return value of an if expression. The `if` special form can therefore be used to control what gets executed, or to return a value, or both. It's up to you.

The uniformity of value returning means that we never have to explicitly use a return statement, so Scheme doesn't have them. Suppose we wanted to write a function `min` to return the minimum of two numbers. In C, we might do it this way:

```
int min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}
```

In Scheme, we can just do this:

```
(define (min a b)
   (if (< a b)
       a
       b))
```

Whichever branch is taken, the value of the appropriate variable (`a` or `b`) will be returned as the value of that branch of the `if`, which is returned as the value of the whole `if` expression, and that is returned as the return value of the procedure call.

Of course, you can also write a one-branch if, with no "else" clause.

[1] "Ternary" just means "takes three arguments," the ternary operator in C is called that because it's the *only* ternary operator in C; all the others take fewer than three arguments.

```
(if (some-test)
    (some-action))
```

The return value of a one-branch `if` is unspecified in the case the condition is false, so if you're interested in the return value, you should use a two-branch `if`, and explicitly specify what should be returned in both cases.

Notice that the flow of control is top-down, through the nesting of expressions—`if` controls which of its subexpressions is evaluated, which is like the nesting of control statements in most languages. Values flow back up from expressions to their callers, which is like the nesting of expressions in most languages.

You can write an expression that is an ordered sequence of other expressions, using `begin`. For example,

```
(begin (foo)
       (bar))
```

calls `foo` and then calls `bar`. In terms of control flow, a `(begin` ... `)` expression is rather like a `begin` ... `end` block in Pascal, or a `{` ... `}` block in C. (We don't need an `end` keyword, because the closing parenthesis does the job.)

Scheme `begin` expressions aren't just code blocks, though, because they are expressions that return a value. A `begin` returns the value of the last expression in the sequence. For example, the `begin` expression above returns the value returned by the call to `bar`.

The bodies of procedures work like `begin`s as well. If the body contains several expressions, they are evaluated in order, and the last value is returned as the value of the procedure call.

Here's a procedure `baz` that calls `foo` and then calls `bar` and returns the result from the call to `bar`.

```
(define (baz)
   (foo)
   (bar))
```

### 2.2.3 The Boolean Values `#t` and `#f`

Scheme provides a special unique object, whose written representation is `#f`, called *false*. This object counts as false if it's the result of a condition expression in an `if` (or `cond`) expression. In most Schemes, this is the *only* value that counts as false, and all others count as true.

The false object is *not* the same thing as the integer zero (as it is in C), and it's not the same thing as a null pointer (as it is in Lisp). The false object is a unique object.

For convenience and clarity, Scheme also provides another boolean value, written `#t`, which can be used as a true value. Note that in general, any value other than false is true, but the special boolean object `#t` is a good one to use when all you want to say is that something is true—returning the true boolean makes it clear that all you're returning is a true value, not some other value that conveys more information.

Like other objects, Booleans are conceptually objects on the heap, and when you write `#t` or `#f`, it means "a pointer to the canonical true object" or "a pointer to the false object."

Scheme provides a few procedures and special forms for operation on booleans. The procedure `not` acts as a `not` operator, and always returns true or false (`#t` or `#f`). If applied to `#f`, it returns `#t`. Since all other values count as true, applying `not` to anything else returns `#f`.

### 2.2.4 Some Other Control-Flow Constructs: `cond`, `and`, and `or`

We've already seen that the special form `if` is a kind of expression, which returns a value as well as affecting control flow. Scheme also has `cond`, a more general conditional construct, and `and` and `or`. These are all value-returning expressions; they're also special forms, not procedures: they control *whether* expressions get evaluated, depending on the values returned by other expressions.

#### 2.2.4.1 `cond`

In most procedural programming languages, you can write a sequence of `if` tests using a an extended version of `if`, something like this:

```
if test1 then
   action1();
else if test2 then
   action2();
else if test3 then
   action3();
else
   action4();
```

Scheme has a similar construct, a special form called `cond`. The above example might be written in Scheme as

```
(cond (test1
        (action1))
       (test2
        (action2))
       (test3
        (action3))
       (else
        (action4)))
```

Notice that each test-and-action pair is enclosed in parentheses. In this example, `test1` is just a variable reference, not a procedure call, i.e., we're testing to see if the value of the variable `test1` is `#f`; if not, we'll execute `(action1)`, i.e., call the procedure `action1`. If it is false, control "falls through" to the next test, and keeps going until one of the tests evaluates to a true value (anything but `#f`).

Notice that we indent the actions corresponding to a test by one character. This lines the actions up directly under the tests, rather than under the opening parenthesis that groups them together.

The `else` clause of a `cond` is optional; if present, that branch will be taken "by default"—if none of the other conditions evaluates to a true value, the `else` branch will be taken.

We don't really need the else clause, because we could get the same effect by using a test expression that always evaluates to a true value. One way of doing this is to use the literal `#t`, the true boolean, because it's always true.

```
(cond (test1
        (action1))
       (test2
        (action2))
       (test3
        (action3))
       (#t             ; literal #t is always true, so
        (action4)))    ; this branch is taken if we get this far
```

The code above is equivalent to a nested set of `if` expressions:

```
(if test1
    (action1)
    (if test2
        (action2)
        (if test3
            (action3)
            (if #t
                (action4)))))
```

Like an `if`, a `cond` returns the value of whatever "branch" it executes. If `test1` is true, for example, the above `cond` will return the value returned from the procedure call `(action1)`.

Remember that each branch of an `if` is a *single* expression; if you want to execute more than one expression in a branch, you have to wrap the expressions in a `begin`. With `cond`, you don't have to do this. You can follow a test expression with more than one action expression, and Scheme will evaluate all of them, in order, and return the value of the last one, just like a `begin` or a procedure body.

Suppose we want to modify the above `cond` example so that it prints out the branch it's taking, as well as evaluating the action expression and returning its value. We can do this:

```
(cond (test1
       (display "taking first branch")
       (action1))
      (test2
       (display "taking second branch")
       (action2))
      (test3
       (display "taking third branch")
       (action3))
      (else
       (display "taking fourth (default) branch")
       (action4)))
```

This `cond` will return the same value as the original, because it always returns the value of the last expression in a branch. As it executes, however, it also displays what it's doing. We can use the `cond` both for *value* and for *effect*.

Be particularly careful about parentheses with `cond`. You *must* enclose each branch with a pair of parentheses around the test expression and the corresponding sequence of action expressions. If you want to call a procedure in any of those expressions, you must *also* put parentheses around the procedure call. In the above example, if we wanted the first test to be a call to a procedure `test1`—rather than just fetching the value of the variable `test1`—we'd write

```
(cond ((test1)
       (display "taking first branch")
       (action1))
      ...)
```

instead of

```
(cond (test1
       (display "taking first branch")
       (action1))
      ...)
```

(Note the indenting here. We usually line up a test and the corresponding sequence of actions vertically, whether or not the expression starts with a parentheses. That is, we indent one space past the opening parenthesis of the pair of parentheses that goes around them all.)

Don't be afraid to use `cond` for conditionals with only one or two branches. `cond` is often more convenient than `if` because it can execute a sequence of expressions, instead of just one. It's not uncommon to see things like this:

```
...
(cond ((foo)
       (bar)
       (baz)))
...
```

Don't be confused by this—there's only one branch to this `cond`, like a one-branch `if`. We could have written it

```
...
(if (foo)
    (begin (bar)
           (baz)))
...
```

It's just more convenient to use `cond` so that we can call `bar` before calling `baz` and returning its result, without explicitly writing a `begin` expression to sequence them.

We say that `cond` is *syntactic sugar* for nested `if`s with `begin`s around the branches. There's nothing we can do with `cond` that we can't do straightforwardly with `if` and `begin`—`cond` just gives us a "sweetened" syntax, i.e., one that's more convenient.

Most of the special forms in Scheme are like this—they're just a convenient way of writing things that you *could* write using more basic special forms. (There are only five "core" special forms that are really necessary, and the others are equivalent to combinations of those special forms.)

#### 2.2.4.2 `and` and `or`

The special forms `and` and `or` can be used as logical operators, but they can also be used as control structures, which is why they are special forms.

`and` takes any number of expressions, and evaluates them in sequence, until one of them returns `#f` or all of them have been evaluated. At the point where one returns `#f`, `and` returns that value as the value of the `and` expression. If none of them returns `#f`, it returns the value of the last subexpression.

This is really a control construct, not just a logical operator, because whether subexpressions get evaluated depends on the reults of the previous subexpressions.

`and` is often used to express both control flow and value returning, like a sequence of `if` tests. You can write something like

```
(and (try-first-thing)
     (try-second-thing)
     (try-third-thing))
```

If the three calls all return true values, `and` returns the value of the last one. If any of them returns `#f`, however, none of the rest are evaluated, and `#f` is returned as the value of the overall expression.

Likewise, `or` takes any number of arguments, and returns the value of the first one that returns a true value (i.e., anything but `#f`). It stops when it gets a true value, and returns it without evaluating the remaining subexpressions.

```
(or (try-first-thing)
    (try-second-thing)
    (try-third-thing))
```

`or` keeps trying subexpressions until one of them does return a true value; if that happens, `or` stops and returns that value. If none of them returns anything but `#f`, it returns `#f`.

```
=====================================================================
This is the end of Hunk A.

TIME TO TRY IT OUT

At this point, you should go read Hunk B of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
=====================================================================
```

(Go to Hunk B, which starts at Section 3.1 [Interactive Prog Envt], page 83.)

### 2.2.5 Comments (Hunk C)

```
==================================================================
Hunk C starts here:
==================================================================
```

[ I should say this earlier: ]

You can and should put comments in your Scheme programs. Start a comment with a semicolon. Scheme will ignore any characters after that on a line. (This is like the `//` comments in C`++`.)

For example, here's a variable definition with a comment after it:

```
(define foo 22) ; define foo with an initial value of 22
```

Of course, most comments should tell you things that *aren't* patently obvious from looking at the code.

Standard Scheme does not have block comments like C's `/*...*/` comments.

### 2.2.6 A Note about Parentheses and Indenting

The two biggest barriers to learning Scheme are probably parentheses and indenting. In Scheme, parentheses are used a little differently than in most programming languages. Indenting is also very important, because the surface syntax of the language is so regular. When reading Scheme code, experienced programmers read the indenting structure as much as the tokens. If you don't parenthesize correctly, your programs won't run correctly. And if you don't indent them correctly, they'll be hard to understand.

The syntax of Scheme is more similar to that of C or Pascal than it may appear at first glance. After all, almost all programming languages are based on nested (statements or) expressions. Like C or Pascal, Scheme is free-form, and you can indent it any way you want.

Some people write Scheme code indented like C, with closing parentheses lined up under opening parentheses to show nesting. (People who do this are usually beginners who haven't learned to use an editor properly, as I'll explain later.) They might write

```
(if a
    (if b
        c
        d
    )
    e
)
```

rather than

```
(if a
    (if b
        c
        d)
    e))
```

The first version looks a little more like C, but it's not really easier to read. The second example shows its structure just as clearly if you know how to read Scheme, and is in fact easier to read because it's not all stretched out. The second example takes up less space on the page or a computer screen. (This is important when editing code in a window and doing other things in another window—you can see more of your program at a time.)

There are a couple of things to keep in mind about parentheses in Scheme. The first thing is that *parentheses are significant*. In C or Pascal, you can often leave parentheses out, because of "operator precedence parsing," where the compiler figures out the grouping. More importantly, you can often add extra parentheses around expressions without affecting their meanings.

This is not true in Scheme! In Scheme, the parentheses are not just there to clarify the association of operators. In Scheme, parentheses are not optional, and *putting extra parentheses around things changes their meaning*. For example, the expression `foo` is a variable reference, whose effect is to fetch the value of the variable `foo`. On the other hand, the expression `(foo)` is a call to the procedure named `foo` with zero arguments.

(Notice that even in C, it's not generally acceptable to write a procedure call with too few parentheses or too many: a call `foo(a, b)` can't be written just `foo a, b` or as `foo((a, b))`.)

In general, you have to know where parentheses are needed and where they are not, which requires understanding Scheme's rules. Some parentheses indicate procedure calls, while others are just delimiters of special forms. Luckily, the rules are simple; they should become very clear in the next chapter or two.

The other thing to know about parentheses is that they have to match. For every opening parenthesis there has to be a closing parenthesis, and of course it must be in the right place.

#### 2.2.6.1 Let Your Editor Help You

Matching parentheses is easy if you have a decent text editor. For example, in `vi`, you can position the cursor over a parenthesis and hit `%`, and it will scan forward or backward (from an opening or closing parenthesis, respectively) to find the matching parenthesis and highlight it, skipping over any matched parenthesis pairs; it will warn you if no match is found.

Most editors have a feature like this. Learn to use it. It's usually easy to get the opening parentheses right, and then if you're in doubt, use the editor to make sure you get the closing parentheses in the right place.

Some editors, like Emacs, have special modes for editing Lisp and Scheme. This can be helpful, but just helping match parentheses is the crucial thing for an editor for Scheme. One of the nice things about the Emacs Scheme mode is that it will indent your code automatically if you like, which will show you whether your expressions nest the way they think you do—if you don't get the parentheses right, the text will look funny and tip you off to your error.

(One Emacs mode for Scheme is `cmuscheme`, which is available from the usual sources of Emacs mode code. It's just a set of Emacs Lisp routines that customizes Emacs to "understand" Scheme syntax and help you format it. You use the Emacs Lisp package `cmuscheme.el`, and it gives you a handy Scheme editing mode. It's available from the Scheme Repository.)

Even without a special package, an editor can help you a lot. For example, most modes in Emacs automatically match parentheses, flashing an opening parentheses when you type the corresponding closing parenthesis. A few minutes figuring out how your editor matches parentheses will save you a lot of time.

#### 2.2.6.2 Indenting Procedure Calls and Simple Control Constructs

#### 2.2.6.3 Indenting `cond`

Be careful about parentheses and indenting with `cond`. Notice that the expressions within a test-action clause are indented by only one character, but that's very significant. Without that indenting, a `cond` is very hard to read.

```
(if (a)
    (begin (b)
           (c))
    (begin (e)
           (f)))
```

We could write it like this:

```
(cond ((a)
       (b)
       (c))
      (else
       (e)
       (f)))
```

Sometimes, when the clauses of a cond are small, a whole clause will be written out horizontally. The above example is likely to be written like this:

```
(cond ((a) (b) (c))
      (else (d) (e)))
```

Also be careful about the parentheses around condition expressions. Notice that the parentheses around `(a)` are there because the condition is call to `a` with zero arguments, not because you always put parentheses around the condition expression. (Notice that there are no parentheses around `#t`, and there wouldn't be parentheses around `a` if we just wanted to test the value of the variable `a`, rather than call it and test the result.)

#### 2.2.6.4 Indenting Procedure Definitions

As I hinted earlier, there's a special rule for indenting procedure definitions. You generally indent the body of a procedure a few characters (I use 3), but you *don't* line the body expressions up directly under the list of variable names.

Don't do this:

```
(define (double x)
        (+ x x))
```

If you do this, a procedure definition looks like looks like a procedure call, or a normal variable definition. To make it clearer you're defining a procedure, do this:

```
(define (double x)
   (+ x x))
```

This makes it clear that the `(double x)` is a different kind of thing from `(+ x x)`. The former declares how the procedure can be called, and the latter says what it will do.

### 2.2.7 All Values are Pointers to Objects

As I said earlier, all values are conceptually pointers to *objects* on a heap, and you don't ever have to explicitly free memory.

By "object," I don't necessarily mean object in the object-oriented sense. I just mean *data objects* like Pascal records or C structs, which can be referenced via pointers and may (or may not) hold state information.

*Some* versions of Scheme do have object systems for object-oriented programming. (This includes our own RScheme system, where standard Scheme types are all classes in a unified object system.) In this book, however, we use the word "object" in a broader sense, meaning an entity that you can have a pointer to.

#### 2.2.7.1 All Values are Pointers

[ some of this needs to be moved up: ]

Conceptually, all Scheme objects are allocated on the heap, and referred to via pointers. This actually makes life simple, because you don't have to worry about whether you should dereference a pointer when you want to use a value—you always do. Since pointer dereferencing is uniform, procedures *always* dereference a pointer to a value when they really use the value, and you never have to explicitly force the dereferencing.

For example, the predefined Scheme procedure `+` takes two pointers to numbers, and automatically dereferences both pointers before doing the addition. It returns a pointer to the number that's the result of the addition.

So when we evaluate the expression `(+ 2 3)` to add two to three, we are taking a pointer to the integer `2` and a pointer to integer `3`, and passing those as arguments to the procedure `+`. `+` returns a pointer to the integer `5`. We can nest expressions, e.g., `(* (+ 2 3) 6)`, so that the pointer to five is passed, in turn, to the procedure `*`. Since these functions all accept pointers as arguments and return pointers as values, you can just ignore the pointers, and write arithmetic expressions the way you would in any other language.

When you think about it, it doesn't make any sense to change the value of an integer, in a mathematical sense. For example, what would it mean to change the integer `6`'s value to be `7`? It wouldn't mean anything sensible, for sure. `6` is a unique, abstract mathematical object that doesn't have any state that can be changed—`6` is `6`, and behaves like `6`, forever.

What's going on in conventional programming languages is *not* really changing the value of an integer—it's replacing one (copy of an) integer value with (a copy of) another. That's because most programming languages have both pointer semantics (for pointer variables) and *value* semantics (for nonpointer variables, like integers). You make multiple copies of values, and then clobber the copies when you perform an assignment.

In Scheme, we don't need to clobber the value of an integer, because we get the effect we want by replacing pointers with other pointers. An integer in Scheme is a unique entity, just as it is in mathematics. We don't have multiple copies of a particular number, just multiple references to it. (Actually, Scheme's treatment of numbers is not quite this simple and pretty, for efficiency reasons I'll explain later, but it's close.)

As we'll see later, an implementation is free to optimize away these pointers if it doesn't affect the programmer's view of things—but when you're trying to understand a program, you should always think of values as pointers to objects.

The uniform use of pointers makes lots of things simpler. In C or Pascal, you have to be careful whether you're dealing with a raw value or a pointer. If you have a pointer and you need the actual value, you have to explictly dereference the pointer (e.g., with C's prefix operator `*`, or Pascal's postfix operator `^`). If you have a value and you need a pointer to it, you have to take its address (e.g., with C's prefix `&` operator, or Pascal's prefix operator `^`).

In Scheme, none of that mess is necessary. User-defined routines pass pointers around, consistently, and when they bottom out into predefined routines (like the built-in `+` procedure or `set!` special form) those low-level built-in operations do any dereferencing that's necessary.

(Of course, when traversing lists and the like, the programmer has to ask for pointers to be dereferenced, but from the programmer's point of view, that just means grabbing another pointer value out of a field of an object you already have a pointer to.)

It is sometimes said that languages like Scheme (and Lisp, Smalltalk, Eiffel, and Java) "don't have pointers." It's at least as reasonable to say that the opposite is true—everything's a pointer. What they don't have is a distinction between pointers and nonpointers that you have to worry about.[2]

#### 2.2.7.2 Implementations Optimize Away Pointers

You might think that making every value a pointer to an object would be expensive, because you'd have to have space for all of the pointers as well as the things they point to, and you'd have to use extra instructions to access things via pointers.

Everything's a pointer at the language level—i.e., from the programmer's point of view—but a Scheme system doesn't actually have to represent things the way they appear at the languages level.

Most Scheme implementations optimize away a lot of pointers. For example, it's inefficient to actually represent integer values as pointers to integer objects on the heap. Scheme implementations therefore use tricks to represent integers without really using pointers. (Again, keep in mind that this is just an implementation trick that's hidden from the programmer. Integer values have the semantics of pointers, even if they're represented differently from other things.)

Rather than putting integer values on the heap, and then passing around pointers to them, most implementations put the actual integer bit pattern directly into variables—after all, a reasonable-sized integer will fit in a machine word.

[2] This also has to do with what you mean by "pointer." I use the word to mean pointers in the sense of building pointer-linked data structures. (Scheme clearly has those.) Some people use "pointers" to mean *addresses* that are bit patterns you can manipulate directly—the way you can in C, where you can "cast" (coerce) a pointer to an integer and operate on the bits. Some people use "pointer" synonymously with "address", and call what Scheme has *object references*.

A short value (like a normal integer) stored directly into a variable is called an *immediate value*, in contrast to pointers which are used to refer to objects indirectly.

The problem with putting integers or other short values into variables is that Scheme has to tell them apart from each other, and from pointers which might have the same bit patterns.

The solution to this is *tagging*. The value in each variable actually has a few bits devoted to a *type tag* which says what kind of thing it is—e.g., whether it's a pointer or not. The use of a few bits for a tag slightly reduces the amount of storage available for the actual value, but as we'll see next, that usually isn't a problem.

It might seem that storing integer bit patterns directly in variables would break the abstraction that Scheme is supposed to present—the illusion that all values are pointers to objects on the heap. That's not so, though, because the language enforces restrictions that keep programmers from seeing the difference.

In the case of numbers and a few other types, you can't change the state of the object itself. There's no way to side-effect an integer object and make it behave differently. We say that integers are *immutable*, i.e., you can't *mutate* (change) them.

If integers were actually allocated on the heap and referred to via pointers, and if you *could* change the integer's value, then that change would be visible through other pointers to the integer.

(That doesn't mean that a variable's value can't be one integer at one time, and another integer at another—the variable's value is really a pointer to an integer, not the integer itself, and you're really just replacing a pointer to one integer with a pointer to another integer.)

### 2.2.7.3 Objects on the Heap

Most Scheme objects only have fields that are general-purpose *value cells*—any field can hold any Scheme value, whether it's a tagged immediate value or a tagged pointer to another heap-allocated object. (Of course, conceptually they're all pointers, so the type of a field is just "pointer to anything.")

So, for example, a *pair* (also known in Lisp terminology as a "cons cell") is a heap-allocated object with two fields. Either field can hold any kind of value, such as a number, a text character, a boolean, or a pointer to another heap object.

The first field of a pair is called the `car` field, and the second field is called the `cdr` field. These are among the dumbest names for anything in all of computer science. (They are just a historical artifact of the first Lisp implementation and the machine it ran on.)

Pairs can be created using the procedure `cons`. For example, to create a pair with the number 22 as the value of its `car` field, and the number 15 as the value of its `cdr` field, you can write the procedure call `(cons 22 15)`.

The fields of a pair are like variable bindings, in that they can hold any kind of Scheme value. Both bindings and fields are called *value cells*—i.e., they're places you can put any kind of value.

In most implementations, each heap-allocated object has a hidden "header" field that you, as a Scheme programmer, are not supposed to know about. This extra field holds type information, saying exactly what kind of heap allocated object it is. So, laid out in memory, the pair looks something like this:

```
        +-----------+
  header| <PAIR-ID> |
        +===========+
     car|     +-----+----->22
        +-----------+
     cdr|     +-----+----->15
        +-----------+
```

In this case, the `car` field of the pair (cons cell) holds the integer `22`, and the cdr field holds the integer `15`.

The values stored in the fields of the pair are drawn as arrows, because they are pointers to the numbers 22 and 15.

(The actual representation of these values might be a 30-bit binary number with a two-bit tag field used to distinguish integers from real pointers, but you don't have to worry about that.)

Scheme provides a built-in procedure `car` to get the value of the car field of a pair, and `set-car!` to set that field's value. Likewise there are functions `cdr` and `set-cdr!` to get and set the `cdr` field's values.

Suppose we have a top-level variable binding for the variable `foo`, and its value is a pointer to the above pair. We would draw that situation something like this:

```
                             +---------+
    +---------+        header| <PAIR>  |
foo |    *----+------------->+=========+
    +---------+           car|    *----+---->22
                             +---------+
                          cdr|    *----+---->15
                             +---------+
```

Most other objects in Scheme are represented similarly. For example, a vector (one-dimensional array) is typically represented as a linear array of value cells, which can hold any kind of value.

Even objects that aren't actually represented like this can be *thought of* this way, since conceptually, everything's on the heap and referred to via a pointer.

### 2.2.8 Scheme Reclaims Memory Automatically

In languages like C or Pascal, data objects may be allocated in several ways. (Recall that by "objects" I just mean data objects like records.) They may be allocated statically (as in the case of global variables), or on an activation stack as part of a procedure activation record (as in the case of local variables), or dynamically allocated on the heap at run time using an alloction routine like `malloc` or `new`.

Scheme is simpler—all objects are allocated on the heap, and referred to via pointers. The Scheme heap is *garbage collected*, meaning that the Scheme system automatically cleans up after you. Every now and then, the system figures out which objects aren't in use anymore, and reclaims their storage. (This determination is very conservative and safe—the collector will never take back any object that your program holds a pointer to, or might reach via any path of pointer traversals. Don't be afraid that the collector will eat objects you still care about while you're not looking!)

The use of garbage collection supports the abstraction of *indefinite extent*. That means that all objects conceptually live forever, or at least as long as they might matter to the program—there's no concept (at the language level) of reusing memory. From the point of view of a running program, memory is infinite—it can keep allocating objects indefinitely, without ever reusing their space.

Of course, this abstraction breaks down if there really isn't enough memory for what you're trying to do. If you really try to create data structures that are bigger than the available memory, you'll run out. Garbage collection can't give you memory you don't have.

Some people think that garbage collection is expensive in time and/or space. While garbage collection is not free, it is much cheaper than is generally believed. Some people have also had bad experiences with systems that stop for significant periods to collect garbage, but modern GC's can solve this problem, too. (If you're interested in how efficient and nondisruptive garbage collectors are implemented, a good place to start is my GC survey paper, available from my research group's web site at `http://www.cs.utexas.edu/users/oops`.)

### 2.2.9 Objects Have Types, Variables Don't

If I use my finger as a pointer, I can use it to point to all kinds of things—a computer, a painting, a motorcycle, or any number of things. Variables in Scheme are like this, too.

#### 2.2.9.1 Dynamic typing

In Scheme, all variables have the same type: "pointer to anything."

Scheme is *dynamically typed*, meaning that variables don't have fixed types, *but objects do.* An object carries its type around with it—an integer is an integer forever, but a variable may refer to an integer at some times, and a string (or something else) at other times. The language provides type-checking at run time to ensure that you don't perform the wrong operations on objects—if you attempt to add two strings, for example, the system will detect the error and notify you.

Sometimes, people refer to languages like Scheme (and Lisp and Smalltalk) as *untyped.* This is very misleading. In a truly untyped language (like FORTH and most assembly languages), you can interpret a value any way you want—as an integer, a pointer, or whatever. (You can also do this in C, using unsafe casts, which is a source of many time-consuming bugs.[3] )

[3] An unsafe cast is one that the compiler doesn't really understand. A safe cast is one that makes sense to the compiler, such as converting an integer to a floating point number. With unsafe casts, you're essentially telling the compiler "trust me", and bypassing the type systems. This is what happens when you cast a structure pointer to a `void*` or `char*` and later cast it back to a structure pointer—you're promising the compiler that the pointer will actually point to a structure of the declared type, and it's up to you to make sure that's true.

In dynamically typed systems, types are enforced at runtime. If you try to use the numeric procedure `+` to add two lists together, for example, the system will detect the error and halt gracefully—it won't blithely assume you know what you're doing and corrupt your data. You also can't misinterpret a nonpointer value as a pointer, and generate fatal segmentation violations that kill your program.

You might think that dynamic typing is expensive, and it can be. But good Scheme compilers can remove most of the overhead by inference at compile time, and most advanced implementations also let you declare types in performance-critical places so that the compiler can generate code similar to that for C or Pascal.

[ I've left out some text from my course notes about tagging and immediate values (more detailed)... put back in, maybe in an appendix ]

```
==================================================================
This is the end of Hunk C

TIME TO TRY IT OUT

At this point, you should go read Hunk D of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk D, which starts at Section 3.1.10 [Making Some Objects], page 95.)

### 2.2.10 The Empty List (Hunk E)

```
==================================================================
Hunk E starts here:
==================================================================
```

In Scheme, there is one null pointer value, called "the empty list," which prints as `()`. (Later, we'll see why it's written that way, and why it's called "the empty list.")

Conceptually, the empty list is a special object, and a null pointer is a pointer to this special end-of-list object. You can ignore that fact and think of it as just a null pointer, because there's nothing interesting you can do with the object it points to.

(In some implementations, the empty list object `'()` is actually an object referred to via a pointer, and null pointers are really pointers to it. In others, an empty list is an immediate value, a specially tagged null pointer. At the level of the Scheme language, it doesn't matter which way it's implemented in a particular Scheme system. All you can really do with the null pointer is compare it against other pointers, to see if they're null pointers, too.)

The empty list object acts as a null pointer for any purpose—there's only one kind of pointer (pointer to anything), so there's only one kind of null pointer (pointer to nothing).

Scheme provides a procedure, `null?` to check whether a value is (a pointer to) the empty list, i.e., a null pointer. For example, `(null? foo)` returns `#t` if the value of the variable `foo` is the empty list, and `#f` otherwise.

You might be wondering why the null pointer object is called "the empty list"; I'll explain that later. Given the way lists are usually used in Scheme, it turns out to make perfect sense.

You can write the empty list as a literal in your programs as `'()`. That is, the expression `'()` returns the empty list (null pointer), `()`. Later I'll explain why you have to put the single quote mark in front of the empty set of parentheses when writing the empty list as a literal.

## 2.3 Pairs and Lists

Scheme, like Lisp, has built-in procedures for dealing with a particularly flexible kind of list—a list of pairs, whose `cdr` fields hold pointers that string them together, and whose `car` fields hold the values. (That is, the `cdr` fields act as "next" pointers, linking the pairs into a linear list.)

### 2.3.1 `cdr`-linked lists

In Lisp and Scheme, you don't typically string objects together into a list by giving each one a "next" field that points right to the next object. Instead, you create a list of pairs whose `car` fields hold the pointers to the objects, and whose `cdr` fields link the pairs together into a "spine."

There isn't really a special `list` data type in Scheme. A list is really just a sequence of pairs, ending with a null pointer. A null pointer is a list, too—it's a sequence of *zero* pairs ending in a null pointer. We sometimes talk about "the car of a list" or "the cdr of a list," but what that really means is "the car of the first pair in the list" and "the cdr of the first pair in the list."

Suppose we have a variable `foo` holding a pointer to a list containing the integers `22`, `15`, and `6`. Here's one way of drawing this situation.

```
                   +---------+          +---------+        +---------+
    +---------+    | <PAIR>  |          | <PAIR>  |        | <PAIR>  |
foo |    *----+--->+=========+      +-->+=========+    +-->+=========+
    +---------+    |       22|     /    |       15|   /    |        6|
                   +---------+    /     +---------+  /     +---------+
                   |    *----+---+      |    *----+-+      |    *    |
                   +---------+          +---------+        +---------+
```

This shows something pretty close to the way things are likely to actually represented in memory. But there's usually a better way of drawing the list, which emphasizes the fact that number values are conceptually pointers to numbers, and which corresponds to the way we usually think about lists:

```
    +---+    +---+---+      +---+---+      +---+---+
bar | *-+--->| * | *-+----->| * | *-+----->| * | * |
    +---+    +-+-+---+      +-+-+---+      +-+-+---+
               |              |              |
              \|/            \|/            \|/
              22             15              6
```

I've left off the header fields of objects, which are not visible to a Scheme programmer.

I've also drawn pairs in a special way, with the `car` and `cdr` fields side-by-side. I've drawn the integers outside the pairs, with pointers to them from the `car` fields, because that's the way things look at the language level.

This emphasizes the fact that lists are generally separate things from the items "in" the list.

A major advantage of this is that you don't have to modify an object to put it on a list—an object can easily be in many lists at once, because a list is really just a spine of pairs that holds *pointers to* the items in the list. This is much cleaner than the way people are typically taught to create simple lists in most beginning programming classes. (It's also very natural in a language where all values are pointers—*of course* lists of objects are really just lists of pointers to objects.)

For example, you can have two lists with the same elements, or some of the same elements, but perhaps in a different order.

```
        +---+    +---+---+      +---+---+      +---+---+
    bar | *-+--->| * | *-+----->| * | *-+----->| * | * |
        +---+    +-+-+---+      +-+-+---+      +-+-+---+
                   |              |              |
                  \|/            \|/            \|/
                  22             15              6
                  /|\                           /|\
                   |                             |
        +---+    +-|-+---+                     +-+-+---+
    baz | *-+--->| * | *-+-------------------->| * | * |
        +---+    +---+---+                     +---+---+
```

Here I've drawn two lists, `bar` and `baz`—that is, lists that are the values of the variables `bar` and `baz`. `bar` holds the elements `22`, `15`, and `6`, while `baz` just holds the elements `22` and `6`.

Since these two lists are really just made up of pairs, and they're *different* pairs, we can modify one list without modifying the other, and without modifying the objects "in" the lists. For example, we can reverse the order of one of the lists without affecting the other.

(We also don't have to create a special kind of list node that has two next fields, so that something can be in two lists at a time. We can just have two separate lists of pairs, or three or four.)

Scheme has a standard way of writing a textual representation of a list. Given the pictured situation, evaluating the expression `(display bar)` will print `(22 15 6)`. Evaluating the expression `(display baz)` will print `(22 6)`. Notice that Scheme just writes out a pair of parentheses around the items in the list—it doesn't represent the individual pairs, but just their `car` values.

Dynamic typing also helps make lists useful. A list of pairs can hold any type of object, or even a mixed bag of different types of objects. So, for example, a pair list can be a list of integers, a list of lists, a list of text characters, or a list of any of the kinds of objects we haven't gotten to yet. It can also be a mixed list of integers, other lists, and whatnot. A few list routines can therefore be useful in a variety of situations—a single list search routine can search any kind of list for a particular target object, for example.

This picture shows two variable bindings, for the variables `bar` and `foo`. `bar`'s binding holds a list `(10 15 6)`, while `foo`'s holds a list `(22 15 6)`. We say that these lists *share structure*, i.e., part of one list is also part of the other.

```
                  +-------- +
    +---------+    | <PAIR>  |
bar |    *----+--->+=========+
    +---------+    |       10|
                  +---------+
                  |    *----+-+
                  +---------+  \
                                \
                  +---------+    \     +---------+       +---------+
    +---------+   | <PAIR>  |     \    | <PAIR>  |       | <PAIR>  |
foo |    *----+--->+=========+     +-->+=========+   +-->+=========+
    +---------+   |       22|     /    |       15|   /   |        6|
                  +---------+    /     +---------+  /    +---------+
                  |    *----+---+      |    *----+-+     |    *    |
                  +---------+          +---------+       +---------+
```

This picture may correspond well to how things are represented in memory, but it's a little confusing.

The more common way of drawing this data structure is

```
    +---+    +---+---+
bar | *-+--->| * | *-+------+
    +---+    +-+-+---+      |
               |            |
              \|/           |
              10            |
                           \|/
    +---+    +---+---+      +---+---+      +---+---+
foo | *-+--->| * | *-+----->| * | *-+----->| * | * |
    +---+    +-+-+---+      +-+-+---+      +-+-+---+
               |              |              |
              \|/            \|/            \|/
              22             15             6
```

Again, this emphasizes the idea that everything's a pointer—conceptually, the pairs hold pointers to the integers.

Generally, pairs are drawn as pairs of little boxes, and they're typically drawn with the boxes side by side—that's just handy because pairs are often used for linear lists, which you want to display horizontally—it's easy to draw the spine of the list horizontally if the `cdr` field (used as a "next" pointer) is on the right. (Of course, when there's shared structure, as in the above picture, you can't draw all cdrs going directly to the right.)

We leave off the headers because they're a low-level detail anyway, because they're a hidden implementation detail that may vary from system to system, and because Scheme programmers immediately recognize this kind of two-box drawing of a pair.

In the above picture, we can talk about "the car of `foo`", which really means the value in the `car` field of the pair pointed at by the value stored in (the binding of) `foo`. It's (a pointer to) `10`. We would often call this "the car of the list `foo`."

Notice that the cdr of `foo` is also a list, and it's *the same list* as the cdr of `bar`—the cdrs of the first pairs in each list point to the same list.

We can say that the cdr of foo and the cdr of bar "are `eq?`," because the expression `(eq? (cdr foo) (cdr bar))` returns true. That is, `(car foo)` and `(cdr foo)` return (pointers to) exactly the same object.

### 2.3.2 Lists and Quoting

Scheme allows you to write lists as literals using *quoting*. Just as you can write a literal boolean or number in your program, you can write a literal list if you use the special form `quote`.

For example, the expression `(quote (1 2 3))` returns a pointer to a list `(1 2 3)`, i.e., a sequence of cdr linked pairs whose car values are (pointers to) to 1, 2, and 3.

You can use `quote` expressions as subexpressions of other expressions, because they just return pointer values like anything else.

For example, the expression `(define foo (quote (1 2 3)))` defines (and binds) a variable `foo`, and initializes its binding with (a pointer to) a three-element list.

We can draw the resulting situation this way:

```
     +---+    +---+---+      +---+---+      +---+---+
foo  | *-+--->| * | *-+----->| * | *-+----->| * | * |
     +---+    +-+-+---+      +-+-+---+      +-+-+---+
                |              |              |
               \|/            \|/            \|/
                1              2              3
```

`quote` takes exactly one argument, and returns a data structure whose printed representation is the same as what you typed in as the argument to `quote`. Scheme does *not* evaluate the argument to `quote` as an expression—it just gives you a pointer to a data structure.

Note that `quote` does not generally construct a character string—it constructs a data structure that may be a list or tree or even an array. It's a very general quoting facility, much more powerful than the double quotes around character strings, which only construct string objects.

Scheme provides a cleaner way of writing `quote`d expressions, using the special single-quote character `'`. Rather than writing out `(quote` *some-expression*,`)` you can just precede the quoted expression with the single-quote character. For example, we can write the same definition of `foo` as `(define foo '(1 2 3))`. You don't need a closing quote, because of Scheme's parenthesized prefix syntax—it can figure out where the quoted data structure ends.

### 2.3.3 Where the Empty List Got its Name

Now that you understand Scheme lists and simple quoting, I can explain why the null pointer is called "the empty list," and written `'()`.

Consider a list `foo` of three elements:

```
(define foo '(1 2 3))
```

The `cdr` of that list is a list `(2 3)`. We could write a literal list like that as `'(2 3)`

The `cdr` of *that* list is a one-element list, `(3)`. We could write a literal list like that as `'(3)`.

The cdr of *that* list is a zero-element list, (), that is, it's the empty list. We could write it in quoted form as '().

Given the way that Scheme lists work, a list of zero items is the same thing as a null pointer, and it's natural to for Scheme to print it as a list with zero elements, ()—and for you to write it as a literal with a single quote, '().

```
==================================================================
This is the end of Hunk E.

TIME TO TRY IT OUT

At this point, you should go read Hunk F of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk F, which starts at Section 3.1.11 [Lists], page 101.)

[ Maybe I should introduce strings and symbols here, moving some material from the tutorial chapter here and possibly expanding the tutorial with more examples. ]

## 2.4 Type and Equality Predicates (Hunk G)

```
==================================================================
Hunk G starts here:
==================================================================
```

Since a pointer can point to any kind of thing, it's often good to know what kind of thing it does point to. For example, you might have a mixed list of different kinds of things, and want to go through the list, doing a different operation for each kind of object you encounter. For this, Scheme provides *type predicates*, which are procedures which test to see whether the pointed-to object is of a particular type.

You also often want to know whether two values refer to the same object, or to data structures with the same structure. For this, Scheme provides *equality predicates*.

These procedures are called "predicates" because they test whether a property is true of a value, and return a yes-or-no answer—that is, the boolean #t or the boolean #f. (This is like a "predicate" in formal logic, which is a kind of statement with a "truth value" that depends on its arguments.)

The names of predicates generally end with a question mark, to signify that they return a boolean. When you write your own programs, it's good style to end the names of boolean-valued (true/false) functions with a question mark.

(An exception to this rule is the standard numeric comparison predicates like `<`, `>`, and `=`. By the rule, they should have question marks after their names, but they're used very frequently and people generally recognize that they're predicates. We don't bother with question marks in their names, because it would clutter up arithmetic expressions.)

### 2.4.1 Type Predicates

Scheme provides built-in procedures to test whether values refer to objects of particular types. If you want to know whether the value of variable `x` is (a pointer to) pair, you can use the predicate `pair?`, like this: `(pair? x)`.

Likewise, if you want to know if something is a number, you can use the predicate `number?`. If you want to know whether a value is an integer, and not just some kind of number, you can use `integer?`.

Several other type predicates are provided, for other data types we'll discuss later, including `string?`, `character?`, `vector?`, and `port?`.

### 2.4.2 Equality Predicates

Equality predicates tell whether one value is "the same as" another. There are actually several important senses of "the same as," so Scheme provides four equality predicates.

Sometimes you want to know whether two data structures are structurally the same, with the same values in the same places. For example, you may want to know whether a list has the same structure and elements as another list. For this, you can use `equal?`, which does a deep, element-by-element structural comparison.

For example `(equal? '(1 2 3) '(1 2 3))` returns `#t`, because the arguments are both lists containing `1`, `2`, `3`, in that order. `equal?` does a deep traversal of the data structure, so you can hand it nested lists and other fairly complicated data structures as well. (Don't hand it structures with directed cycles of pointers, though, because it may loop forever without finding the end.)

`equal?` works to compare simple things, too. For example, `(equal? 22 22)` returns `#t`, and `(equal? #t 15)` returns `#f`. (Note that `equal?` can be used to compare things that may or may not be of the same type, but if they're not, the answer will always be `#f`. Objects of different types are never `equal?`.)

Often you don't want to structurally compare two whole data structures—you just want to know if they're the *exact same object*. For example, given two pointers to lists, you may want to know if they're pointers to the very same list, not just two lists with the same elements.

For this, you use `eq?`. `eq?` compares two values to see if they refer to the same object. Since all values in Scheme are (conceptually) pointers, this is just a pointer comparison, so `eq?` is always fast.

(You might think that tagged immediate representations would require `eq?` to be slower than a simple pointer comparision, because it would have to check whether things were really pointers. This isn't actually true—`eq?` just compares the bit patterns without worrying whether they represent pointers or immediates.)

Equality tests for numbers are treated specially. When comparing two values that are supposed to be numbers, `=` is the appropriate predicate. Using `=` has the advantage that using it on non-numbers is an error, and Scheme will complain when it happens. If you make a mistake and have a non-number where you intend to have a number, this will often show you the problem. (You could also use `equal?`, but it won't signal an error when applied to non-numbers, and may be a little bit slower.)

There is another equality predicate, `eqv?`, which does numeric comparisons on numbers (like `=`), and identity comparisons (like `eq?`) on anything else.

```
==================================================================
This is the end of Hunk G

TIME TO TRY IT OUT

At this point, you should go read Hunk H of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk H, which starts at Section 3.2 [Using Predicates], page 107.)

### 2.4.3 Choosing Equality Predicates (Hunk I)

```
==================================================================
Hunk I starts here:
==================================================================
```

The reason that the `=` and `eqv?` predicates are needed is that the numeric system of Scheme is not quite as clean as it could be, for efficiency reasons.

Ideally, there would be exactly one copy of any numeric value, and all occurrences of that value would really be pointers to the same unique object. Then you could use `eq?` to compare numbers for identity, just as you can for other kinds of values. (For example, there would be just one floating-point number with the value 2.36529, and any computation that returned that floating-point value would return a pointer to that unique object. `(eq? 2.36529 2.36529)` would return `#t`.)

Unfortunately, for numbers it would be too expensive to do this—it would require keeping a table of all of the numbers in the system, and probing that table to eliminate duplicate copies of the same values. As a concession to efficiency, Scheme allows multiple copies of the same number, and the `=` and `eqv?` predicates mask this wart in the language—they perform numeric comparisons when faced with numbers, so that you don't have to worry about whether two numbers with the same value are literally the same object.

`eqv?` thus tests whether two values are "equivalent," when two objects with the same numeric value are treated as "the same," like `=`, but all other objects are distinguished by their object identity, like `eq?`. In general,

- `eq?` is useful for fast identity (same object) comparisons of non-numbers,
- `=` performs numeric comparisons on numbers,
- `eqv?` is like `eq?`, but treats copies of the same number as though they were the same object, and
- `equal?` performs a "deep" comparison of the structure of data structures. (It uses `eqv?` for components that are numbers.)

## 2.5 Quoting and Literals

Programs often need to refer to literal data values—data that you type directly into the program. In many languages, the only literals are fairly simple values like integers and strings. In Scheme, you can use simple literals or complicated ones that represent (pointers to) data structures like nested lists. Earlier, I showed how using quoting to create list literals.

You've probably noticed that the syntax of Scheme code and the textual representation of Scheme data are very similar. So, for example, `(min 1 2)` is a combination if it's viewed as code, but it's also the standard textual representation of a list containing the symbol `min` and the integers `1` and `2`.

(A symbol is a data object that's sort of like a string, but with some special properties, which will be explained in the next chapter.)

The resemblance between code and data is no accident, and it can be very convenient, as later examples will show. It can be confusing, too, however, so it's important to know when you're looking at a piece of code and when you're looking at a piece of data.

The first thing to understand is *quoting*. In Scheme, the expression `(min 1 2)` is a procedure call to `min` with the arguments `1` and `2`.

As I explained earlier, we can quote it by wrapping it in the special form `(quote...)`, however, and get a literal list `(min 1 2)`.

For example, the definition

```
(define foo (quote (min 1 2)))
```

defines and binds `foo`, initializing the binding with (a pointer to) the list `(min 1 2)`.

We can draw this situation this way:

```
    +---+    +---+---+     +---+---+     +---+---+
foo | *-+--->| * | *-+---->| * | *-+---->| * | * |
    +---+    +-+-+---+     +-+-+---+     +-+-+---+
               |             |             |
              \|/           \|/           \|/
              min            1             2
```

Of course, as I explained earlier, we can use `'` as a euphemism for `(quote ... )`

We can define very complicated literals this way, if we want to. Here's a procedure that returns a nested list of nested lists of integers and booleans and symbols:

```
(define (fubar)
   '(((1 two #f) (#t 3 four))
     ((five #f 6) (seven 8 #t))
     ((#f 9 10)) ((11 12 #f))))
```

that's a pretty useless procedure, but it's very convenient to just be able to type in printed representations of nested data structures and have Scheme construct them automatically for you. In most languages you'd have to do some fairly tedious hacking to construct a list like that. As we'll see in a later chapter, Scheme also supports *quasiquotation*, which lets you construct *mostly*-literal data structures, and create customized variations on them easily; quasiquotation will be discussed a later chapter.

### 2.5.1 Simple Literals and Self-Evaluation

You might have noticed by now that we've already been using literals a lot in our examples—numeric literals and boolean literals. Why didn't we have to quote them to keep Scheme from trying to evaluate them like other expressions? Because Scheme has a special rule, which is that the value of a number or boolean is that number or boolean. For these data types, the result of attempting to evaluate it is the same as what you started with. So the value of `4` is `4`, and the value of `#f` is `#f`. (This also works for a few other types, such as characters and character strings.) Scheme lets you type in the text representation of a value as an expression, and by convention the value of that expression is the value you typed the printed representation of. Such an expression is called *self-evaluating*, because it is evaluated to itself.

What's the deep meaning of this rule? There isn't any. It's just to keep you from having to type a lot of quotes to use simple literals. Notice that that means that you can quote a number or boolean if you want, and it doesn't make any difference. The expression `'0` means "literally the number 0," but since Scheme defines the value of a number to be itself, the value of plain `0` is `0`, too.

Likewise, the value of `'#f` or `(quote #f` is the same as `#f`—they're all pointers to the false object. You can write a string literal `'"foo"` as `"foo"`. In either case, the value of the expression is a pointer to a string object with the character sequence `f o o`.

Minor warning: don't add extra quotes *inside* expressions that are already quoted. `'(foo 10 baz)` is *not* the same thing as `'('foo '10 'baz)`. One quote for a whole literal expression is enough, and extra quotes inside quotes do something that will seem surprising until you understand how quoting really works.

Expression evaluation in Scheme is simple, for the most part, but you must remember the rules for the special forms (which don't always evaluate their arguments) and self-evaluation. Later, I'll show how an interpreter implements self-evaluation by analyzing expressions before evaluating them. Still later, I'll show how a compiler can do the same work at compile time, so that using literals doesn't cost any evaluation overhead at run time.

## 2.6 Local Variables and Lexical Scope

Scheme is a *block-structured* language with *nested scopes*. You can declare local variables whose scope is a block of code, and blocks can have blocks inside them with their own local variables.

Scheme uses a *lexical scope* rule. (We can also say that Scheme is *statically scoped*, rather than *dynamically scoped*, like some old Lisps.) When you see a variable name in the code, you can tell what variable it refers just to by looking at the source code for the program. A program consists of possibly nested blocks of code, and the meaning of the name is determined by which variable binding constructs it's used inside.

### 2.6.1 `let`

You can create code blocks that have local variables using the `let` special form.

You've seen local binding environments in other languages before. In C or Pascal you've probably seen blocks with local variables of their own, e.g., in C:

```
...
{  int x = 10;
   int y = 20;

   foo(x,y);
}
...
```

Here we've got a block (inside curly braces) where local variables named `x` and `y` are visible. (The same thing can be done with `begin`...`end` blocks in Pascal.)

When we enter the block, storage is allocated for the local variables, and the storage is initialized with the appropriate initial values. We say that the variables are *bound* when we enter the block—the names `x` and `y` refer to something, namely the storage allocated for them. (In C, the storage for local variables may be allocated on an activation stack.)

This is a simple but important idea—when you enter a scope, you "bind" a name to storage, creating an association (naming) between a name and a place you can put a value. (In later chapters, we'll see how interpreters and compilers keep track of the association between names and storage.)

Sometimes, we refer to the storage allocated for a variable as "its binding," but really that's a shorthand for "the storage named by the variable," or "the storage that the variable is bound to."

Inside the block, all references to the variables `x` and `y` refer to these new local variable bindings. When execution reaches the end of the block, these variable bindings cease to exist and references to `x` or `y` will again refer to whatever they did outside the block (perhaps global variables, or block variables of some intermediate-level block, or nothing at all).

In this example, all that happens inside the block is a call to the procedure `foo`, using the values of the block variables, i.e., `10` and `20`. In C or Pascal, these temporary variables might be allocated by growing the stack when the block is entered, and shrinking it again when the block is exited.

In Scheme, things are pretty similar. Blocks can be created with `let` expressions, like so:

```
...
(let ((x 10)
      (y 20))
  (foo x y))
...
```

The first part of the `let` is the variable binding clause, which in this case two subclauses, `(x 10)` and `(y 20)`. This says that the `let` will create a variable named `x` whose initial value is `10`, and another variable `y` whose initial value is `20`. A `let`'s variable binding clause can contain any number of clauses, creating any number of `let` variables. Each subclause is very much like the name and initial value parts of a `define` form.

The rest of the `let` is a sequence of expressions, called the *let body*. The expressions are simply evaluated in order, and the value of the last expression is returned as the value of the whole `let` expression. (The fact that this value is returned is very handy, and will be important in examples we use later.)

A `let` may only bind one variable, but it still needs parentheses around the whole variable binding clause, as well as around the (one) subclause for a particular binding. For example:

```
...
(let ((x 10))
   (foo x))
...
```

(Don't forget the "extra" parentheses around the one variable binding clause—they're not really extra, because they're what tells Scheme where the variable binding clause starts and stops. In this case, before and after the subclause that defines the one variable.)

In Scheme, you can use local variables pretty much the way you do in most languages. When you enter a `let` expression, the `let` variables will be bound and initialized with values. When you exit the `let` expression, those bindings will disappear.

You can also use local variables differently, however, as we'll explain in later chapters. In general, the bindings for Scheme variables aren't allocated on an activation stack, but on the heap. This lets you keep bindings around after the procedure that creates them returns, which will turn out to be useful.

(You might think that this is inefficient, and it could be, but good Scheme compilers can almost always determine that it's not really necessary to put most variables on the heap, and avoid the cost of heap-allocating them. As with good compilers for most languages, most variables are actually in registers when it matters, so that the generated code is fast.)

#### 2.6.1.1 Indenting `let` Expressions

In general, we indent `let` expressions in a way that shows the block structure of the program. The binding forms (variable names and initial values) are lined up vertically after the keyword `let`, and the body expressions are indented a few characters and lined up vertically, like so:

```
(let ((x 10)    ; bindings of x
      (y 20))   ; and y
   (foo x)
   (let ((a (bar))   ; bindings of a
         (b (baz)))  ; and b
      (quux x a)
      (quux y b))
   (baz))
```

Notice that the binding forms of each `let` are lined up vertically, and the `body` expressions are *not* indented as far. This is important for making it obvious where the binding forms stop and the body expressions start. (In this example, the body of the outer `let` consists of a call to `foo`, another `let`, and a call to `baz`. The body of the inner let consists of two calls to `quux`.)

### 2.6.2 Lexical Scope

If nested `let`s define variables by the same name, then the uses of that name in the body of the inner *let* will refer to the bindings created by the inner *let*.

Consider the following code fragment:

```
(let ((x 10)    ; outer binding of x
      (a 20))   ; binding of a
   (foo x)
   (let ((x (bar))       ; inner binding of x
         (b (baz x x)))  ; binding of b
      (quux x a)
      (quux y b))
   (baz x a)  ; refers to outer x (and a)
   (baz x b)) ; illegal?
```

When control enters the outer `let`, the inital values for the variables are computed. In this case, that's just the literal values `10` and `20`. Then storage is allocated for the variables, and initialized with those values. Once that's done, the meaning of the names `x` and `a` changes—they now refer to the new storage for (bindings of) the `let` variables `x` and `a`—and then the body expressions are evaluated.

Similarly, when control enters the inner `let`, the inital values are computed by the calls to `bar` and `baz`, and then storage for `x` and `b` is allocated and initialized with those values. Then the meanings of the names `x` and `b` change, to refer to the new storage (bindings) of those variables. (For example, the value of `x` when `(baz x x)` is evaluated is still 10, because `x` still refers to the outer `x`.)

In this example, the meaning of the identifier `x` changes when we enter the inner `let`. We say that the inner let variable `x` *shadows* the outer one, within the body of the `let`. The outer `x` is no longer visible, because the inner one is.

When we exit a `let` (after evaluating its body expressions), the bindings introduced by the `let` "go out of scope," i.e., aren't visible anymore. (For example, when we evaluate the expression `(baz x a)` in the body of the outer `let`, `x` refers to the binding introduced by the outer `let`—the `x` introduced by the inner let is no longer visible.

Likewise, in the example code fragment, the `b` in the last expression, `(baz x b)`, does not refer to the inner `let`'s binding of `b`. Unless there is a binding of `b` in some outer scope we haven't shown (such as a top-level binding), then this will be an error.

#### 2.6.2.1 Binding Environments and Binding Contours

The set of all bindings that are visible at a given point during program execution is called a *binding environment*. That is, a binding environment maps a set of names to the pieces of storage they name.

A top-level binding environment is the mapping that the Scheme system maintains between top-level variable names and the storage they're bound to. This might be implemented as a hash table.

With local variables, a simple "flat" table isn't sufficient. Entering a `let`, for example, adds new bindings to the environment that code is executing in—it makes the new variable bindings visible, changing the mapping from names to storage.

We say that each binding contruct we execute introduces a new *binding contour*. We call it a contour because it changes the "shape" of the environment.

You can think of a binding contour as being implemented by a new table that's created when you enter a `let`, or any other construct that binds variables. When Scheme looks for a binding of an identifier, it looks first in this new table, then in the old table that represented the environment outside the *let*. Since Scheme looks in the "inner" environment's table first, it will always find the innermost binding of any identifier, such as `x` in the example above.

At any given point, the *environment* consists of *all* of the variable bindings that are visible. This includes all of the bindings in the table for the innermost contour, and all of the bindings in the table for the contours it's nested inside, *except* those that are shadowed by inner bindings of the same names.

#### 2.6.2.2 Block Structure Diagrams for `lets`

We can make environments and contours clearer by drawing a block diagram showing where the different variables are visible:

```
(let ((x 10)     ; bindings of x
      (a 20))    ; and a
 +-----------------------------------------------------------+
 | (foo x)                                  scope of outer x |
 | (let ((x (bar))                            and a          |
 |       (b (baz x x)))                                      |
 |   +-------------------------------------------------+     |
 |   | (quux x a)                    scope of inner x  |     |
 |   | (quux y b)                    and b             | )   |
 |   +-------------------------------------------------+     |
 | (baz x a)                                                 |
 | (baz x b)                                                 | )
 +-----------------------------------------------------------+
```

(This kind of block diagram is the origin of the term "block structure.")

Each box represents a contour: it shows where in the program each variable binding will be visible.

We can interpret a block structure diagram by looking outward from an occurrence of a variable name, and using the nearest enclosing box that corresponds to a binding of that name. Now we can see that the final call `(baz x b)` does not refer to the `let` variable `b`—it's not inside the box corresponding to that variable. We can also see that the occurrence of `x` in that expression refers to the outer `x`. The occurrence of `x` in the calls to `quux` refer to the inner `x`, because they're inside its box, and inner definitions shadow outer ones.

There's something a little tricky to notice here. When we evaluate the initial value expressions for the inner `let`, the inner bindings *are not visible yet.* `x` still refers to the *outer* binding of `x`, not the inner one that we are about to create. Sometimes this is exactly what you want, but sometimes it's not. Because it isn't always what you want, Scheme provides two variants of `let`, called `let*` and `letrec`.

### 2.6.3 `let*`

`let` is useful for most local variables, but sometimes you want to create several local variables in sequence, with each variable's value available to compute the next variable's value.

For example, it is common to "destructure" a data structure, extracting part of the structure, then a part of that part, and so on. We could do this by simply nesting expressions that extract parts, but then we don't have understandable names for the intermediate results of the nested expressions.

(In other cases, we may want to do more than one thing with the results of one of the nested expressions, so we need to create a variable so that we can refer to it in more than one body expression.)

Consider the code fragment:

```
(let ((a-structure (some-procedure)))
   (let ((a-substructure (get-some-subpart a-structure)))
      (let ((a-subsubstructure (get-another-subpart a-substructure)))
         (foo a-substructure))))
```

Scheme provides a convenient syntax for this sort of nested let; can be written as a single `let*`

```
(let* ((a-structure (some-procedure))
       (a-substructure (get-some-subpart a-structure))
       (a-subsubstructure (get-another-subpart a-substructure)))
   (foo a-substructure))))
```

Notice that this wouldn't work if we wrote it as a normal `let` that binds three variables. A block structure diagram shows why:

```
(let ((a-structure (some-procedure))
      (a-substructure (get-some-subpart a-structure))
      (a-subsubstructure (get-another-subpart a-substructure)))
 +-----------------------------------------------------------------+
 | (foo a-substructure)          ; scope of all three variables | )))
 +-----------------------------------------------------------------+
```

Now we see that all of the initial value expressions for the `let` variables are outside the scope of any of the variables. `a-substructure` and `a-substructure` will not refer to the bindings introduced by this `let`, but to whatever bindings (if any) are visible outside the `let`.

With `let*`, it's different:

```
(let ((a-structure (some-procedure))
 +---------------------------------------------------------------------+
 |    (a-substructure (get-some-subpart a-structure))                  |
 +------------------------------------------------------------------+  |
 |    (a-subsubstructure (get-another-subpart a-substructure)))   |  |
 +---------------------------------------------------------------+  |  |
 | (foo a-subsubtructure)                                         |  |  | )))
 +---------------------------------------------------------------+--+--+
```

Each initial value clause is in the scope of the previous variable in the `let*`. From the nesting of the boxes, we can see that bindings become visible one at a time, so that the value of a binding can be used in computing the initial value of the next binding.

There's another local binding construct in Scheme, `letrec`, which is used when creating mutually recursive local procedures. I'll discuss that later, when I describe how local procedures work in Scheme.

```
==================================================================
This is the end of Hunk I

TIME TO TRY IT OUT

At this point, you should go read Hunk J of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
```

```
==================================================================
```

Go to Hunk J, which starts at Section 3.3 [Local Variables], page 112.

## 2.7 Procedures (Hunk K)

```
==================================================================
Hunk K starts here:
==================================================================
```

Scheme procedures are "first class," meaning that they're objects in the language. They can be anonymous, meaning that you can have pointers to procedures that don't have printed names. They can be higher-order, meaning that procedures can operate on procedures.

### 2.7.1 Procedures are First Class

In Scheme, procedures are data objects—you can have a pointer to a procedure and do the same things you can do with any other Scheme value. Technically, we say that procedures are *first class* objects in the language—you can pass a procedure value as an argument to a procedure, return it as the value of a procedure call, store it in a variable or a field of another object. A procedure pointer is just a value that you can pass around like any other value, like a pair or a boolean.

Procedures are special, of course, because they're the only kind of object that supports the procedure call operation.

In Scheme terminology, a procedure call expression is called a *combination*. Evaluation of a combination includes evaluation of the argument expressions and *application* of the procedure to the arguments, i.e., actually calling it with ("applying it to") those values.

An unusual feature of Scheme is that it uses a *unified namespace*, which means that there's only one kind of name for both normal variables and procedures—in fact, procedure names are really just variable names, and there's only one kind of variable. A named procedure is really just a first-class procedure object that happens to be referenced from a variable.

When you define a procedure as we did above for the `min` example, you're really doing three things: creating a procedure, creating a normal variable (named `min`), and initializing the variable with a pointer to the procedure.

(This means that you can't have both a procedure variable and a "normal" data variable by the same name in the same scope—there's really only one kind of variable, so you can only have one binding in a given scope.)

When you define a procedure as we did above for the `min` example, Don't let the special syntax for procedure definitions fool you—a procedure name is really just the name of a variable that happens to hold a procedure value. You can use any variable that way, by storing a procedure value in it. You can also assign a new procedure value to a variable, and then use it to name the new procedure.

For example, if you've defined `min` as above, you can change the value in the binding of `min` by saying `(set! min +)`. That assignment expression will look up the value of the variable `+`, which is the addition procedure, and assign that into the variable `min`.

Then when you call `min` as before, it will do addition instead, because it will call the same procedure as `+`. For example `(min 5 10)` will return `15`, not `5`.

You could also change the meaning of `+`, just by assigning a new value to the (the binding of) the variable `+`. This is probably a bad idea unless you really have a good reason, because if the new procedure doesn't do addition, any code that calls `+` will return different answers!

It is important to understand how procedure calls actually work in Scheme, which is actually very simple. Consider the combination (procedure call expression) `(+ a b)`. What this really means is

1. look up the value of (the current binding of) the variable `+`, which we *assume* is a procedure,
2. look up the values of (the current bindings of) the variables `a` and `b`, and
3. apply the procedure to those values, i.e., call it with those values as arguments.

The first subexpression of the combination is evaluated in just the same way as the others, although the result is used differently. The first subexpression is just a subexpression that should return a procedure value, and the others give the arguments to pass to it.

This won't work if the first subexpression doesn't evaluate to a procedure value. For example, you can change the meaning of `+` with an assignment expression `(set! + 3)`. Then if you attempt to call `+` with the combination `(+ 2 3)` you'll get an error. Scheme will say something like "ERROR: Attempt to apply non-procedure."

The fact that the first (operator) subexpression is evaluated just like any other expression can be very useful. Rather than giving the name of a particular procedure to call, we can use any expression whose result is a procedure. For example, we might have a table of procedures to use in different kinds of situations, and search that table for the procedure to call at a particular time:

```
((look-up-appropriate-procedure key) foo bar)
```

Here we call the procedure `look-up-appropriate-procedure` with the argument `key` to get a procedure, and then apply it to the values of `foo` and `bar`.

One warning about combinations: the Scheme language doesn't specify the order in which the subexpressions of a combination are evaluated. Don't write code that depends on whether the operator expression is evaluated first, or on the order of evaluation of the argument expressions.

You might wonder what's so special about first-class procedures, since some other languages (like C) let you pass around pointers to procedures, and call them via those pointers. Scheme's procedures work like Pascal's if you use them for the kinds of things Pascal allows, but also lets you use them in more general ways that I'll explain later.

### 2.7.2 Higher-Order Procedures

Scheme is designed to make it easy to use *higher-order* procedures, i.e., procedures that may take other procedures as arguments or return them as values.

For example, you can easily write a `sort` procedure that takes a comparison procedure as an argument, and uses whatever procedure you hand it to determine the sorted order.

To sort a list in ascending order, you can then call `sort` with (a pointer to) the procedure `<` ("less than") as its argument, like this:

```
(sort < '(5 2 3))
```

and you'll get back a sorted list `(2 3 5)`.

Note that the expression `<` here is just a variable reference. We're fetching the value of the variable `<` and passing it to `sort` as an argument.

If you'd rather sort the list in descending order, you can pass it the procedure `>` ("greater than") instead:

```
(sort > '(5 2 3))
```

and get back a sorted list `(5 3 2)`.

The same procedure can be used with lists of different kinds of objects, as long as you supply a comparison operator that does what you want.

For example, to sort a list of character strings into alphabetic order, you can pass `sort` a pointer to the standard string-comparison procedure `string<?`,

```
(sort string<? '("foo" "bar" "baz" "quux"))
```

and get back a list `("bar" "baz" "foo" "quux")`.

[ give `map` example here? ]

### 2.7.3 Anonymous Procedures and `lambda`

Scheme has a special form that is very special, called `lambda`. It creates a first-class procedure and returns a pointer to it.

For example, you can create a procedure that doubles its argument by evaluating the expression `(lambda (x) (+ x x))`. The second subform of the expression is a list of formal arguments, and the third subform is the body of the procedure.

`lambda` doesn't give a name to the procedure it creates—it just returns a pointer to the procedure object.

Actually, the procedure-defining variant of `define` is exactly equivalent to a variable-defining `define`, with a `lambda` expression as its initial value form.

For example,

```
(define (double x)
   (+ x x))
```

is *exactly* equivalent to

```
(define double (lambda (x)
                 (+ x x)))
```

In either case, we're creating a one-argument procedure, and we're also defining and binding a variable named `double`, and initializing its storage with a pointer to the procedure.

The procedure-defining syntax for `define` is just syntactic sugar—there's nothing you can do with it that you can't do with local variables and `lambda`. It's just a more convenient notation for the same thing.

```
==================================================================
This is the end of Hunk K.

TIME TO TRY IT OUT

At this point, you should go read Hunk L of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk L, which starts at Section 3.4.2 [Using Procedures], page 116.)

### 2.7.4 `lambda` and Lexical Scope (Hunk M)

```
==================================================================
Hunk M starts here:
==================================================================
```

`lambda` creates a procedure that will execute in the scope where the `lambda` expression was evaluated.

Except for local variables of the procedure itself, including its arguments, names in the body of the procedure refer to whatever they refer to at the point where the procedure is created by `lambda`.

This is necessary for preserving lexical scope—the meanings of variable names must be obvious at the point where the procedure is defined.

Local variables created by the procedure have the usual scope rule within the body. (Argument variables are just a special kind of local variable, which get their initial values from the caller.) Other variables are called *free* variables—that is variables defined outside the procedure, but referred to from inside it.

We say that `lambda` creates a *closure*, a procedure whose free variable references are "fixed" at the time the procedure is created. Whenever the procedure references a free variable, it will will refer to the bindings of those variables in the environment where the procedure was created.

Consider the following small program

```
(define foo 1)

(define (baz)
   foo)

(define (quux)
   (let ((foo 6))
      (baz)))

(quux)
```

When `quux` is called, it will bind its local variable `foo` and then call `baz`. When `baz` is called from `quux`, however, it will still see the top level binding of `foo`, whose value is `1`. The result of the call to `baz` will be `1`, and that value will be returned as the value of the call to `quux` as well.

There is a very good reason for this, and it's the rule used by most programming languages. It is important that the meaning of a procedure be fixed where it is defined, rather than having the meaning depend on where it is called from. You want to be able to look at the code, and see that the name `foo` refers to particular variable, namely the one that's visible there, at the top level. You don't want to have to worry about the meaning of the procedure `baz` changing, depending on where it's called from.

A block structure diagram may make this clearer. I'll just show the part for the procedure `baz`:

```
(define (quux)
   (let ((foo 6))
     +---------------------------+
     | (baz)        scope of foo | ))
     +---------------------------+
```

This emphasizes the fact that the local `foo` really is local. The definition of `baz` is not inside the box, so it can't ever see `foo`'s local variable `foo`. (The fact that `baz` is *called* from inside the box doesn't matter.)

Conceptually, the procedure `baz` *returns to the environment where it was created* before it executes, and even before it binds its arguments.

In early Lisps, a different rule was used, called *dynamic scope*. In those Lisps, the call to `baz` *would* see the *most recent* binding of `foo`. In this case, it would see the binding created by `quux` just before the call to `foo`. This led to very inscrutable bugs, because a procedure would work sometimes, and not others, depending on the names of variables bound in *other* procedures.

(Dynamic scoping is generally considered to have been a big mistake, and was fixed in recent versions of Lisp, such as Common Lisp, which were influenced by Scheme.)

### 2.7.5 Local Definitions

Scheme lets you define local procedures, scoped inside other procedures or blocks with local variables. This lets you "hide" procedures that only make sense in a certain context, so that they can only be called in that context.

You can define local procedures using `let` and `lambda`, like this:

```
(define (quadruple x)
   (let ((double (lambda (x)
                   (+ x x))))
      (double (double x))))
```

Here we've defined a procedure named `quadruple`, with a local variable named `double`; its value is a procedure that will double its argument value, created with `lambda`.

Notice that when we call `double` from inside the procedure `quadruple`, we call it by the name `double`, which is really the name of a local variable. That's okay, because there's no difference between variable names and procedure names—a call to a named procedure is *always* a lookup of a variable value followed by a call to the procedure it points to.

Also notice that the inner procedure's argument variable `x` shadows the outer procedure's argument variable `x`. Inside the body of `double`, it refers to `double`'s argument, but outside it doesn't. (The code might be easier to read if we chose different names for the two procedures' arguments, but this is just for illustration.)

As with a top-level definition, we can write a local definition using `define` instead of `let`. For example, we could have written the above procedure as:

```
(define (quadruple x)
   (define (double (x)
              (+ x x))))
   (double (double x))))
```

A local `define` acts a lot like `let` with `lambda`. (Actually, it's exactly like a `letrec` with `lambda`, but we haven't discussed `letrec` yet; we will later.)

There's a restriction on internal `define`s—they must be at the beginning of the procedure body (or the beginning of another body, like a `let` body, *before* the normal executable expressions in the body.

Local procedure definitions follow the normal lexical scope rule, like nested `let`s. For example, in the above example, the formal argument `x` of `double` is local to the body of `double`—it's a different variable `x` than the argument `x` of `quadruple`.

```
(define (quadruple x)
   (define (double (x)
 +--------------------------+
 |           +--------+     |
 |           | (+ x x)|)))  |
 |           +--------+     |
 |  (double (double x))     | ))
 +--------------------------+
```

Here the inner box is the scope of `double`'s argument `x`, and the outer one is the scope of the *variable* `double`.

We could have used a different name for the argument to the local procedure, and it wouldn't change the meaning of either procedure:

```
(define (quadruple x)
   (define (double (y)    ; local defn. of double
              (+ y y)))   ;    body of local procedure
   (double (double x)))   ; body of quadruple
```

On the other hand, since there are no local bindings of `+`, `+` refers to whatever it refers to in the context where `quadruple` is defined. Assuming that `quadruple` is a top-level procedure, not a local procedure in some other scope, `+` refers to the top-level binding of `+`. (Remember that a procedure name is really just a variable name, so the scope rules for variables apply to procedure names too.)

### 2.7.6 Recursive Local Procedures and `letrec`

Using `let` and `lambda` to define local procedures will often work, but generally we use `letrec` rather than `let`, because it supports *recursive* local procedures. (That's why it's called `letrec`—it means `let` with *rec*ursive definitions.)

Suppose we tried to use `let` and `lambda` to define a recursive local procedure:

```
(define (foo x)
   (let ((local-proc (lambda (y)
                        ...
                        (local-proc ...)    ; recursive call?  No.
                        ...)))
      ...
      (local-proc x)
      ...)
```

The problem with this example is that what appears to be a recursive call to `local-proc` from inside `local-proc` actually isn't. Remember that `let` computes the initial values of variables, then initializes all of the variables' storage, and only *then* do any of the bindings become visible—when we enter the body of the `let`. In the example above, that means that the local variable `local-`

`proc` *isn't visible to the* `lambda` *expression.* The procedure created by `lambda` will not see its own name—the name `local-proc` in the body of the procedure will refer to whatever binding of `local-proc` exists in the enclosing environment, if there is one.

A block structure diagram may make this clearer:

```
(define (foo x)
   (let ((local-proc (lambda (y)
                     +--------------------------+
                     | ...               scope  |
                     | (local-proc ...)  of y   |
                     | ...                      | )))
                     +--------------------------+
    +-----------------------------------------+
    | ...                        scope of     |
    | (local-proc x)             local-proc   |
    | ...                                     | )
    +-----------------------------------------+
```

Unlike `let`, `letrec` makes new bindings visible `before` they're initialized. Storage is allocated, and the meanings of names are changed to refer to the new local variable bindings, and *then* the initial values of those variables are computed and the variables are initialized.

For most purposes, this wouldn't make any sense at all—why would you want variable bindings to be visible before they have had their initial values installed? For local procedure definitions, however, it makes perfect sense—we want to use `lambda` to create a procedure that can operate on the variables *later*, when it's called.

`lambda` creates a procedure that will start executing in the scope where the `lambda` expression is evaluated, so we need to make the bindings visible before we evaluate the `lambda` expression.

If we use `letrec` in our example, instead of `let`, it works. The procedure `local-proc` can see the *variable* `local-proc`, so it can call itself by its name.

The block structure diagram looks like this:

```
(define (foo x)       +-----------------------------------+
   (let ((local-proc | (lambda (y)                        |
                      |  +---------------------------+    |
                      |  | ...               scope |      |
                      |  | (local-proc ...)  of y  |      |
                      |  | ...                     | ))) |
                      |  +---------------------------+    |
    +-----------------+                                   |
    | ...                              scope of           |
    | (local-proc x)                   local-proc         |
    | ...)                                                |
    +-----------------------------------------------------+
```

The recursive call to `local-proc` will work, because the call is inside the box that corresponds to the scope of the variable `local-proc`.

`letrec` works for multiple *mutually recursive* local procedures, too. You can define several local procedures that can call each other, like this:

```
(define (my-proc)
   (letrec ((local-proc-1 (lambda ()
                             ...
                             (local-proc-2)
                             ...))
            (local-proc-2 (lambda ()
                             ...
                             (local-proc-1)
                             ...)))
      (local-proc-1))) ; start off mutual recursion by calling local-proc-1
```

A block structure diagram shows that each local procedure definition can see the bindings of the other's names:

```
(define (my-proc)
          +-----------------------------------------------------+
   (letrec ( | (local-proc-1 (lambda ()       scope of local-proc-1 |
          |               ...              and local-proc-2     |
          |               (local-proc-2)                        |
          |               ...))                                 |
          | (local-proc-2 (lambda ()                            |
          |               ...                                   |
          |               (local-proc-1)                        |
    +--------+               ...)))                                |
    | (local-proc-1)                                            | ))
    +-----------------------------------------------------------+
```

You can also define plain variables while you're at it, in the same `letrec`, but `letrec` is mostly interesting for defining local procedures.

When the initial value of a `letrec` variable is not a procedure, you must be careful that the expression does not depend on the values of any of the other `letrec` variables. Like `let`, the order of initialization of the variables is undefined.

For example, the following is illegal:

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

In this case, the attempt to compute `(+ x x)` may fail, because the value of `x` may not have been computed yet. For this example, `let*` would do the job—the second initialization expression needs to see the result of the first, but not vice versa:

```
(let* ((x 2)
       (y (+ x x)))
   ...)
```

Be sure you understand why this is illegal, but the `lambda` expressions in the earlier examples are not. When we create recursive procedures using `letrec` and `lambda`, the `lambda` expressions can be evaluated without actually using the *values* of the bindings they reference. We are creating procedures that *will* use the values in the bindings *when those procedures are called*, but just creating the procedure objects themselves doesn't require the bindings to have values yet. It does require that the bindings exist, because each `lambda` expression creates a procedure that "captures" the currently visible bindings—the procedure remembers what environment it was created in.

### 2.7.7 Multiple `defines` are like a `letrec`

Now that you understand `letrec`, I can explain what `define` really does.

Notice that when you `define` top-level variables and procedures, the procedures you create can refer to other variables in the same top-level environment.

It is as though all of the top-level bindings were created by a single big `letrec`, so that the initial value expressions create procedures that can "see" each others' name bindings. Expressions that aren't definitions make up the "body" of this imaginary `letrec`.

Recall that a procedure-defining `define` is equivalent to a variable-defining `define` with a `lambda` expression to compute its initial value.

The following top-level forms

```
...

(define (foo)
   (... (bar) ...))

(define (bar)
   (... (baz) ...))

(define (baz)
   (... (quux) ...))
...
(foo)
...
```

are therefore equivalent to

```
...

(define foo
        (lambda ()
           (... (bar) ...)))

(define bar
        (lambda ()
           (... (baz) ...)))

(define baz
        (lambda ()
           (... (foo) ...)))
...
(foo)
...
```

When we view top-level `define`s as being implicitly like parts of a `letrec`, the program takes the equivalent form

```
(letrec (...

         (foo (lambda ()
                 (... (bar) ...)))
         (bar (lambda ()
                 (... (baz) ...)))
         (baz (lambda ()
                 (... (foo) ...)))
         ...)
 ...
 (foo)
 ...)
```

(Actually, things are scoped like this, but the initial value expressions of `define`s and the non-definition expressions are evaluated in the order they occurred in the source program. For top-level expressions, you can depend on Scheme executing the executable parts of definitions in the order written.)

Local `define`s work pretty this way, too. A Scheme interpreter or compiler recognizes any `define`s that occur at the beginning of a body as being parts of an implicit *letrec*; the subsequent expressions in the same body are treated as the body of the implicit `letrec`.

So the following procedure

```
(define (my-proc)
   (define (local-proc-1)
      ...)
   (define (local-proc-2)
      ...)
   (local-proc-1)
   (local-proc-1))
```

is equivalent to

```
(define (my-proc)
   (letrec ((local-proc-1 (lambda () ...))
            (local-proc-2 (lambda () ...)))
      (local-proc-1)
      (local-proc-2)))
```

If we "desugar" the outer `define`, too, we get

```
(define my-proc
        (lambda ()
           (letrec ((local-proc-1 (lambda () ...))
                    (local-proc-2 (lambda () ...)))
             (local-proc-1)
             (local-proc-2)))
```

### 2.7.8 Variable Arity: Procedures that Take a Variable Number of Arguments

In Scheme, you can easily write procedures that can take a variable number of arguments. Technically, the number of arguments a procedure accepts is called its *arity*, and we call a procedure that accepts a variable number a *variable arity* procedure.[4]

Often, you write procedures that take a certain number of normal (required) arguments, but can take more. When you pass a procedure more arguments than it requires, Scheme packages up the extra arguments in a list, called a *rest list*.

The syntax for a variable arity procedure declaration is the same as for a fixed-arity procedure declaration, except that the last argument is preceded by a dot (`.`). This last argument will hold the rest list of extra arguments, or `()` if no extra arguments are passed.

The rest list argument is just a normal argument, except that it is initialized in a funny way—with a list of all the actual arguments that followed the required ones. Inside the procedure, you can access it like any other variable, and operate on the list with the normal list-manipulating procedures.

Here's a simple variable-arity procedure, which accepts zero required arguments and any number of other arguments. It simply accepts a rest list of however many arguments are given, and then `display`s that *list* of the arguments it was given.

```
(define (display* . args)
   (display args))
```

Calling this procedure with the combination `(display* 1 2 3)` displays the list `(1 2 3)`.

The syntax for declaring variable arity is a little weird, but it's easier to understand if you're familiar with dot notation for improper lists. Intuitively, what's to the right of the dot in dot notation is the "rest" of the list being written. Here we use a symbol as a formal parameter that symbolizes the "rest" of the "list" of arguments, beyond the required ones.

---

[4] Note that here "variable" just means that it varies, and *arity* means that what's varying is the number of argument *variables*. "Arity" comes from words like "nullary," "unary," "binary," and "ternary"—these mean zero-argument, one-argument, two-argument, and three-argument, respectively. (Sometimes people say things like "1-ary" for unary (one argument), "2-ary" for binary, and "*n*-ary" for "variable arity.")

### 2.7.9 `apply`

The procedure `apply` allows you to call any procedure, and specify a list of values to be passed as arguments. `apply` takes a procedure and a list of values, and then calls the procedure with those values as arguments.

For example, `(apply + '(1 2))` passes the values `1` and `2` to `+`, and is equivalent to `(+ 1 2)`.

You'll seldom need to use `apply`, because normal procedure calling works fine in most situations. Occasionally, though, it is convenient to be able to apply a procedure to a list of values that have already been computed. (I'll show an example in [ chapter 4? ].)

## 2.8 Variable Binding Again

So far, I've sometimes casually talked about variables holding values, but that's not quite right. Variables are bound to storage, and storage holds values.

I've also sometimes casually talked about fetching "the value of a variable," but that's really just a shorthand for fetching the value of the *current binding* of a variable, from the *current environment*.

Consider what happens when we define a variable `foo` with the definition `(define foo 10)`. We can draw the binding of the variable in this way:

```
    +-----+
foo |  *--+---> 10
    +-----+
```

When speaking precisely, we say that the variable `foo` is *bound* to the memory location represented by the box on the left. *Binding* just means making an association between a name and something. (There are several senses of "binding"—it's a very general word—but in this book, I'm generally talking about associating program variables with actual storage.)

For brevity, we refer to the location as the variable's *binding*, but binding is really the relationship between the name and the storage it names.

In Scheme terminology, we talk about "bindings" as distinct from variables, because they are two different things. This is true in most other languages as well (e.g., C and Pascal), but usually people don't make the distinction explicit. They'll refer to a program variable as a variable, but they'll also call the storage allocated for a particular instance of that variable a "variable." Usually, experienced programmers aren't confused by this.

In this book, I try to be a little more precise, because the distinction between variables and bindings is especially important in discussing advanced topics that will come up later. For now, rest assured that there's nothing really unusual here—when I distinguish between variables and bindings, that's applicable to most programming languages, not just Scheme. I'm just giving a name to something you probably already know.

(So far, we haven't seen anything really special about Scheme variables and bindings, except that the values in bindings are always pointers.)

### 2.8.1 Identifiers and Variables

In isolation, a textual identifier (name) such us as `foo` isn't even a variable.

The static scoping structure of a program gives names a certain aspect of meaning, and the dynamic execution of the program gives them more meaning.

In isolation, `foo` doesn't mean anything. Used in a program, it can be the name of a *variable*. At different places in a program, it can be the name of *different* variables, e.g., a toplevel variable, or a local variable in one or more procedures.

In Scheme an identifier such as `foo` may not represent a variable at all. In the `quote` expressions `'foo` and `'(baz foo bar)` it identifies a symbol object, but in an entirely different sense than variable binding. It doesn't name a variable `foo`, or a variable whose binding holds a pointer to `foo`—it is a literal representation of a pointer to the unique symbol object whose printed representation is `foo`.

### 2.8.2 Variables, Bindings and Values

The distinction between variables, bindings, and values is particularly important in Scheme, and important for understanding interpreters and compilers, so I'll take a little time to discuss it with examples.

What is this distinction? Why not just say that the variable holds a value, i.e., why not call the unit of storage a variable? Because that's not right. Consider the following short program.

```
(define (double x)          ; define a procedure that doubles its argument
   (+ x x))
(define (quadruple x)       ; define a procedure that quadruples
   (+ (double x)            ; its argument.
      (double x)))
(define (foo x)             ; define a recursive procedure that calls
   (if (> x 0)              ; itself if its argument is more than 0,
       (foo (- x 1))))      ; handing the recursive call an argument
                            ; that's one less.
```

Notice that we've defined three procedures, `double`, `quadruple`, and `foo`, each of which has a local (argument) variable `x`. (An argument variable is just a local variable that gets its initial value in a special way, passed from the caller of the procedure.)

There are therefore *three different variables named* `x` in this code. In each of the procedures, it means something different. Each procedure defines a different meaning for the name `x`, and each separate meaning is a different variable.

This becomes obvious when we talk about "the variable `x` defined in the procedure `double`" versus "the variable `x` in the procedure `foo`," and so on.

Notice also that when we define the procedures, there is no storage allocated for their local variables; the variables named `x` in the procedures are just definitions—no space will be allocated for them until the procedures are actually *called*. That's when binding happens—some storage is allocated at run time to hold the value.

(Bear in mind that this happens in other languages too, even if people don't discuss it clearly—for example, a C argument variable is bound when you enter the procedure, because suddenly space is allocated for it and the name refers to that space.)

When we call something a "variable," that's *not* because we can assign to it and change its value. None of the above variables has a value that varies in that sense; none of these procedures happens to modify the values they're given as arguments. In some languages, such as pure functional languages, you can't do assignment at all, but those languages still have variables.

In programming language terminology, the term "variable" means pretty much what it means in mathematics—at different times we invoke the same procedure and the variable will refer to something different. For example, I may call `double` with the argument `10`, and while executing in that call to `double`, the value of `x` will be `10`. Later, I may call `double` with the value `500`, and while executing in *that* call the value of `x` will be `500`.

Consider how similar this is to variables in logic. I may have a logical statement that "for all *x*, if *x* is a person then *x* is mortal". (Forall x, person(x) -> mortal(x)). I can use the same logical rule (statement) and apply it to lots of things.

If Socrates is a person then Socrates is mortal, and if Bill Clinton is a person then Bill Clinton is mortal, and so on. (Or even, if my car is human then my car is mortal.)

Each time I use it, *x* may refer to a different thing, and that's why it's called a *variable*.

Just because it's a variable doesn't mean that when I use it I change the state of the thing I use it to refer to—for example, Bill Clinton is probably not modified much by the fact that I'm inferring something about him, and I'm pretty sure Socrates isn't changed much at all by the experience.

It also doesn't mean that the meaning of a variable changes from instant to instant. If I use the rule above, and apply it to Socrates, saying "if Socrates is a person then Socrates is mortal", *x* consistently refers to Socrates—that's the point. But I can also say that "if Bill Clinton is a person then Bill Clinton is mortal." In that case *x* refers consistently to Bill Clinton. In logic, we say that in one case *x* is *bound to* Socrates, but then used consistently within the rule; and in the other, we say it's bound to Bill Clinton, and then used consistently within the rule.

The point here is that the same variable can refer to different things at different times. These different things are called *bindings*, because the variable is associated with ("bound to") something different each time.

Consider the recursive procedure `foo`, above. In a recursive procedure, the same variable may be bound to *different* things at the *same time*. Suppose I call `foo` with the argument `15`, and it binds its argument `x` and gives the binding the initial value `15`. Then it examines that value, and calls itself with the argument `14`. The recursive call binds its argument `x` with the value `14`, then examines that and calls itself with the value `13`, and so on.

At each recursive call, a new *binding* of `x` is created, even if the old bindings still exist, because the earlier calls haven't finished yet—they're suspended for the duration of the recursion.

When there are multiple bindings in existence at the same time, only one one is "visible" as a procedure executes. For example, in a recursive set of calls to a procedure, only one binding is "in scope," that is, visible) to an executing procedure—the binding for that call. We call this the *current binding* of the variable. When a call returns, an older binding becomes visible again, and becomes the current binding.

But what is a variable bound to, i.e., to what does a variable refer? In Scheme, it refers to a piece of *storage*. When you call a procedure, for example, each argument variable is bound to a piece of storage that can hold the argument value you pass. Inside that call to that procedure, that variable name will refer to that piece of memory.

A single binding of a Scheme variable may hold different values over time, because of assignments, as in most procedural languages. So not only may the same variable be bound to different pieces of storage, but each piece of storage may hold different values over time.[5]

Sometimes people talk about binding a variable to a *value*, but in Scheme (and other languages with assignment) this is not correct, and speaking in this sloppy way causes confusion. If you don't distinguish between storage and values, you can't talk clearly about assignment.

Always remember that there are *three* "one-to-many mappings" here:

- a single name (identifier) can be used for different variables at different places in a program, or for a symbol
- a given variable may be bound to different pieces of storage, e.g., at different calls to the same procedure,
- a given binding may hold different values if you assign to it and change what's stored there.

To keep these terms straight, it's usually best to think about local variables; top-level or global variables are a special case, because they only have one binding each.

Top-level defines can be a little confusing in terms of the variable/binding/value distinction, because they do *three different* things. They declare a variable that will be visible in a scope (the top level scope), they bind they variable to new storage (creating the top-level binding), and they initialize that storage with an initial value.

---

[5] Here the analogy may not be very intuitive; it's as though you used a name like `x` to refer to different "people" at different times, but *those people were all alike*, and uninteresting except for what you call them and *what they point at* with their index fingers. In this analogy, `set!` is like telling one of these very boring people "point at that for now."

```
==================================================================
This is the end of Hunk M.

TIME TO TRY IT OUT

At this point, you should go read Hunk N of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk N, which starts at Section 3.5.3 [Interactively Changing a Program], page 123.)

## 2.9 Tail Recursion (Hunk O)

```
==================================================================
Hunk O starts here:
==================================================================
```

Many Scheme programs rely heavily on recursion, and Scheme makes it easy to use recursion in ways that aren't feasible in most other languages. In particular, you can write *recursive* procedures which call themselves instead of looping.

When a procedure calls itself in a way that is equivalent to iterating a loop, Scheme automatically "optimizes" it so that it doesn't need extra stack space. You can use recursion anywhere you could use a loop in a conventional language. Technically, loop-like recursion is called *tail recursion*, which we'll explain in detail later.

(The basic idea is that you never have to return to a procedure if all that procedure will do is return the same value ot *its* caller. Scheme can implement such *tail calls* as a kind of GOTO that passes arguments, without saving the state of the caller.)

Some compilers for languages such as C perform a limited form of "tail call optimization," but Scheme's treatment of tail calls, is more general, and standardized, so you can use recursion more freely in your programs, without fear of stack overflow.

And of course, you can use recursion the way you would in most languages, as well as for loops, so recursion can do *both* jobs. While Scheme has conventional-looking looping constructs, they're defined in terms of recursion.

## 2.10 Macros

Scheme is very procedure-oriented, but procedures can't do everything, at least not in a way that is syntactically pretty and efficient.

Sometimes you want to define your own control structures and data-defining expressions that can't be clearly and efficiently expressed as procedures, and for this Scheme provides a *syntactic extension* or *macro* facility.

With *macros*, you can define stereotyped pieces of code, and how to transform them for different purposes.

You might have had bad experiences with macros in other languages, like C, but Scheme's macro system is special. It's an extremely powerful mechanism for abstracting over programs and putting things together in special ways.

As we'll see in a later chapter, with Scheme macros you can effectively reprogram the compiler to change the language and its implementation. This is not something you'll need to do often—most of the time you'll do fine with normal programming and higher-order procedures—but sometimes it's extremely useful for building your own extended version of Scheme to solve particular kinds of problems, or for automating tedious and repetitive aspects of program construction.

## 2.11 Continuations

Scheme has the usual control constructs that most languages have—conditionals (if statements), loops, and recursion—but it also has a very special control structure called `call-with-current-continuation`.

(Warning: `call-with-current-continuation` is *weird*.)

`call-with-current-continuation` allows you to save the state of a computation, package it up as a data structure, and go off and do something else. Whenever you want, you can restore the old saved state, abandoning the current computation and and picking up where the saved computation left off.

This is far more powerful than normal procedure calling and returning, and allows you to implement advanced control structures such as backtracking, cooperative multitasking, and custom exception-handling.

You won't use `call-with-current-continuation` most of the time, because more conventional control structures are usually sufficient. But if you need to customize Scheme with a special control structure to solve a particular kind of problem, you can do it with `call-with-current-continuation`.

```
==================================================================
This is the end of Hunk O.

TIME TO TRY IT OUT

At this point, you should go read Hunk P of the next chapter
and work through the examples using a running Scheme system.
Then return here and resume this chapter.
==================================================================
```

(Go to Hunk P, which starts at Section 3.7.6.2 [Basic Programming Examples], page 143.)

## 2.12 Iteration Constructs

You may have noticed that we haven't discussed iteration constructs much. Scheme does include iteration constructs like `do`, which we'll describe later, but you'll use them less than in most other languages. It's usually easier to use recursion, once you get the hang of it. When you do use iteration constructs, you should also understand that they're really syntactic sugar for recursion.

For most purposes, you can use Scheme's iteration constructs as you would in other languages, but they're actually interestingly different. Scheme's iteration constructs are really syntactic sugar for tail recursion. Anything you can do iteratively, you can do with recursion, and recursion lets you do other things that normal iteration doesn't.

The main difference between Scheme's iteration constructs and the ones you may be used to is that loop variables aren't updated at each iteration. This doesn't mean you don't have loop variables—the difference is that loop variables are *rebound* at each iteration (tail call), rather than being bound once on entry to the loop, and updated (assigned to) at each iteration.

(Don't worry if this doesn't make sense yet—it will later.)

It turns out that having a new binding of the loop variable at each iteration is very convenient when using first-class procedures and continuations. For example, if you create a first-class procedure in a loop body, it can continue to refer to the variable binding for the iteration of the loop that created it.

## 2.13 Discussion and Review

# 3 Using Scheme (A Tutorial)

In this chapter, I'll describe how Scheme works from the user's point of view. You should follow along, experimenting with the actual Scheme system you use.

This chapter is not meant to be read independently of the previous chapter. I've included notes saying which parts of the previous chapter you should read before working through parts of this one. If you haven't already, you should read the first part of that chapter.

This chapter is also not meant to be read without a running Scheme system to try things out.

(If you haven't read Hunk A of the previous chapter, please go to Section 2.1 [What is Scheme?], page 9 and read until you reach the end of Hunk A and are directed back here.)

## 3.1 An Interactive Programming Environment (Hunk B)

```
==================================================================
This is the beginning of Hunk B.
==================================================================
```

Most Scheme implementations are *interactive*. The Scheme system is just a program with a command interpreter. When it starts up, it presents you with a prompt, letting you type in an expression. The Scheme system "interprets" that expression, and does what it says to do. Then it prints out a textual representation of the result of the expression.

(Your Scheme system may have a graphical user interface, but the basic idea is the same—you tell Scheme what to do, and it obediently does it, tells you what happened, and asks for the next command. With a GUI, you may be able to tell Scheme what to do by clicking on buttons, etc.)

This is very similar to an operating system's command interpreter or "shell." A shell is just an interpreter for a language—usually a really ugly language.

The nice thing about an interactive programming environment is that your program doesn't go away after you run it. You're "inside" the program, and you can tell it what to do, but instead of just running to completion, it comes back and asks you what to do next.

The values of variables are still around, and you can look at them if you want to. This makes it easy to debug a program. You can type in definitions of variables and procedures, and then run a procedure and see if it does what you expect. If not, you can redefine it. In effect, you're inside *your* program, and the Scheme system acts as a dispatcher, executing whatever part you want and letting you examine the results. This makes it easy to build and test your program in small pieces, and gradually build up larger and larger pieces that use those pieces.

In this section, we'll go through a simple example session with Scheme, fairly slowly, starting with examples similar to the ones in the previous chapter. I'll assume Scheme is already properly installed on your system. If it's not, you need to get Scheme and install it, or have someone install it for you.

(Plug: you might want to use our Scheme, RScheme, which is free. There are other implementations of Scheme of course, including commercial products and other free implementations. If you're using a different Scheme, its operation should be very similar—see the manual for your system.)

It's a very good idea to follow along with this text in front of a running Scheme system, so that you get used to using it interactively. I'll assume you are doing this, and say "do this" and "do that." You don't have to do it, of course, but it's the best way to learn Scheme.

### 3.1.1 Starting Scheme

First we start up Scheme. If we're using RScheme under UNIX, that's probably by typing `rs` at the UNIX `%` prompt. (RScheme might be installed under a different name on your system, perhaps `rscheme`, if so, use that name instead. If you're not using UNIX, start up RScheme the way you start up any program on your system, perhaps by clicking on its icon. If you're using UNIX but your shell has a different prompt, like `>`, don't worry about it.

```
%rs
```

Now the Scheme system starts up and prints out some information about itself, usually including including the name and version version number, and then gives you a Scheme prompt. We'll pretend that the prompt is `Scheme>`, but on your system it's probably something different. (For RScheme, it's something like `top[0]=>`, where the first few characters give you some information about the state of the system, and the `=>` tells you it's ready for input.)

Scheme then waits for you to type in an expression and hit `<RETURN>`. (By that I mean hit the "RETURN" or "ENTER" key on the keyboard. In some Scheme systems, these may be distinct keys, and you may have to hit "ENTER"; the documentation for your system will tell you which key does what.)

Scheme lets you type, echoing the characters to the screen, and doesn't do anything else until you hit `<RETURN>`. Until you hit `<RETURN>`, you can back up to correct typing mistakes (just as you can in an operating system's command shell), using the delete or backspace key.

Now type in a variable definition `(define myvar 10)`, and hit `<RETURN>`. What's happening on the screen looks something like this.

```
Generational Real-Time Garbage Collector Version 0.5
RScheme version 0.7
Scheme>(define myvar 10)
#void
Scheme>
```

Here we defined a variable named `myvar`, giving it the initial value `10`. Scheme read what we typed and figured out what it meant, and then allocated some storage for the variable binding, and initialized that storage with (a pointer to) `10`. Scheme keeps track of the fact that the storage it allocated is now known as `myvar`, as well as keeping track of the value in it.

What Scheme prints out after evaluating this expression may be different on your system (you may not see `#void`). That's because the Scheme standard doesn't specify what's returned as the value of a definition expression. (It's possible that your Scheme system will print out something a little more verbose, or different, or nothing at all as the value of a `define` expression. Don't worry about it.)

You don't usually use the result value of a definition—you're just defining something to use later. Depending on the implementation you're using, you'll see whatever the implementors chose to have definitions return. In some systems, a special unusable value is returned, and Scheme will suppress the printing of these meaningless values to avoid clutter on the screen.

### 3.1.2 Making mistakes and recovering from them

Sometimes you'll make mistakes when interacting with Scheme. This is quite normal, and if you've done it already, don't worry. When Scheme detects that something's wrong, it will complain. In most text-based Scheme systems, it will give you a special kind of prompt, so that you can type in commands to fix the mistake. In other systems, it may invoke a debugger, which is a program for diagnosing and fixing mistakes. For now, you need to know the command for your system that tells Scheme to give up on trying to fix the mistake, and go back to its normal "top level" interaction mode. Later, you should learn how to use the debugging facilities of your system, but for now just being able to get back to the normal Scheme prompt will do.

Assuming you've looked up the command for aborting an expression (by reading the manual, or asking a help system), you should try it out. You should make a mistake intentionally, watch what the system does, and make sure you can recover from your mistakes.

Here's a good mistake, and a hypothetical response from the Scheme, and a recovery to the normal Scheme prompt. Try this on your system, and make sure you can do the equivalent things:

```
Scheme>(2 3 4)
ERROR: attempt to apply non-procedure 2
break[1]>,toplevel
Scheme>
```

Here, we typed in the expression `(2 3 4)`, which is illegal. The Scheme system recognized it as a compound expression that's not a special form, so it attempted to interpret it as a procedure call, and apply the result of the first subexpression to the results of the other subexpressions. In this case, the first subexpression is `2`, which evaluates to `2`, which isn't a procedure at all. At that point, Scheme complained, telling us we'd tried to use `2` as a procedure, and switched to a "break loop" for debugging.

The break loop presented the special debugging prompt `break[1]>`, asking what to do about it. We typed in the special command `,toplevel` to tell it to go back to normal interaction, and it did, presenting us with a fresh `Scheme>` prompt.

In your system, the prompts and commands are likely to be different. (For example, special commands may start with a colon, rather than a comma, and have different names.) Whatever they are, they'll be simple, and you should learn to use them as soon as possible. See the documentation for your system.

Here's another common mistake, which you will make pretty soon, so you should try it and see what happens and how to get out of it:

```
Scheme>a-variable
ERROR: unbound variable: a-variable
break[1]>,toplevel
Scheme>
```

Here what happened is that we asked Scheme to evaluate the expression `a-variable`. Since `a-variable` is just a normal identifier, like a variable name, Scheme assumed it was supposed to be a variable name, and that we were asking for its value. There wasn't a variable named `a-variable`, though, so Scheme complained. In Scheme terminology for giving a piece of memory a name, we hadn't defined that variable and "bound" it to storage. Scheme couldn't find any storage by that name, much less fetch its value.

(Your system may let you get away with using `set!` on an undefined variable, silently creating a binding automatically. This is not required by the Scheme standard, and programs generally should *not* do this.)

As before, we used the special escape command to abort the attempt to evaluate this broken expression, and get back to normal interaction with Scheme.

### 3.1.3 Returns and Parentheses

A common mistake in Scheme is to forget the closing parentheses of expressions. If you forget a closing parentheses—usually because you need several to close nested expressions—most systems will just hang, waiting you to finish typing the expression.

This is a feature, not a bug. It lets you put `<RETURN>`s (line breaks) in your input, to format the code on the screen as you type it in. When you type in the last closing expression and hit `<RETURN>` again, Scheme recognizes that you've typed in a whole expression, and evaluates it and prints the result.

So if you type in an expression and hit `<RETURN>`, and Scheme doesn't do anything, check to see if you closed all of the parentheses you opened. If not, just type in the missing parenthesis and hit `<RETURN>` again.

(It's also possible that in your system, you have to do something special to get Scheme to evaluate an expression—like hitting a different key, or clicking on a button or a menu item. In such systems, `<return>` may be only for formatting the text you're inputting, and another key tells Scheme to go ahead and evaluate what you've typed.)

### 3.1.4 Interrupting Scheme

Inevitably, you'll sometimes code routines that get stuck in infinite loops (or infinite recursion). You need to know how to stop such loops and get back to the normal Scheme interaction prompt. Scheme systems generally allow you to "interrupt" what the system is doing, and get a new prompt.

In most UNIX-based Scheme systems, you can use `<ctrl>-C`, i.e., hold down the CONTROL key and hit the `c` key, to send an interrupt. In other systems, there will be another keyboard command or a button or menu item you can click. Find out what the command is for your system. You'll need it.

In general, if the system hangs, you should check to see if you closed all of the parentheses you opened—it may just be waiting for you to finish your input. If that doesn't work, and you think the program is stuck in an infinite loop, or some other computation you don't want to wait for, interrupt it with `<CTRL>-C` or the equivalent on your system.

It's possible that even this won't work. After all, Scheme systems can have bugs, too. In very unusual circumstances, you may have to kill the Scheme program more brutally. If you're using a window system, you may be able to just kill the window Scheme is running in. Under UNIX, you can use the `ps` command to figure out the process ID of the Scheme process, and kill it with the `kill` command. (This may require the `-9` option.)

### 3.1.5 Exiting (Quitting) Scheme

When you're through using Scheme interactively, you need to be able to get out of it. You give a command to tell the interactive Scheme system (which is just a program) to terminate.

Most systems have a special command (starting with comma or whatever the convention is), like `,exit`. (It might also be `,quit`, `,halt`, or `,bye`.) There may be a Scheme procedure you can evaluate to kill the system, by evaluating a procedure call expression in the normal way, e.g., `(exit)`, `(halt)`, `(quit)`, or `(bye)`.

In many systems (especially under UNIX), you can use an interrupt key sequence to kill the system, if you're at the top-level. E.g., at the top-level prompt, `<ctrl>-D`, may do it.

### 3.1.6 Trying Out More Expressions

Now that you're familiar with typing in erroneous expressions, let's get back to trying legal ones.

If you've exited your Scheme system, fire it up again.

Type in the addition expression `(+ 2 3)`, and hit `<return>`. (From now on, I'll skip saying "and hit `<return>`." I'll also stop showing the prompt Scheme gives you after printing the result of an expression.)

```
Scheme>(+ 2 3)
5
```

Again, Scheme evaluated the expression, printed the result, which was (a pointer to) `5`, and gave you a prompt to type in something else. Notice that it didn't save the value anywhere. It just printed out the result.

The value we gave to `myvar` earlier is still there, though. We can ask Scheme what it is, just by typing in a variable reference expression, i.e., just the variable name.

```
Scheme>myvar
10
```

Scheme has kept track of the storage named `myvar`, and it evaluates the expression `myvar` by looking up the value. Then it prints out that result, and gives you another prompt, as it always does.

To change the value stored in the binding of `myvar`, and look at the new value, just type in a `set!` expression and then the name of the variable, like this:

```
Scheme>(set! myvar 32)
#void
Scheme>myvar
32
```

You may see a different result for the `set!` expression. Standard Scheme doesn't specify the return value of `set!`, because you generally use it for its side-effect, not its result. As with `define`, your system may return something different. It may also suppress the printing of this useless value, so you may not see anything at all.

In some Scheme systems, the value of a `set!` expression is the name of the variable being set, so you may see somthing like this:

```
Scheme>(set! myvar 32)
myvar
Scheme>myvar
32
```

(In other systems, it's something else, like the old value of the variable you're clobbering.) You should not depend on the value returned by the `set!` if you want your program to be portable. In the example above, it doesn't really matter what result the `set!` returns, except that that's what gets printed out before you get a new prompt. What matters about `set!` is its *effect*, which is to update the value of the variable binding. As we can see, it had its effect—when we evaluate the expression `myvar`, it returns the new value, which is printed out: `32`.

We can also use more complicated expressions—just about anything. Now we'll increment the variable by five, and again ask Scheme the value of the variable.

```
Scheme>(set! myvar (+ myvar 5))
#void
Scheme>myvar
37
```

Now let's define a procedure that expects a number as its argument, and returns a number that's twice as big. Then we'll call it with the argument `2`.

```
Scheme>(define (double x) (+ x x))
#void
Scheme>(double 2)
4
```

After evaluating the first expression, Scheme keeps track of the definition of `double`. When we type in the second expression, Scheme calls that procedure, which returns a result, which Scheme prints out.

Since Scheme keeps track of the variables and values we typed in earlier, we can call `double` to double the value of myvar:

```
Scheme>(double myvar)
74
```

We can define new procedures in terms of old ones. (Actually, we did this when we defined `double`—it's defined in terms of `+`, which is predefined, i.e., Scheme knows that definition when it starts up.)

```
Scheme>(define (quadruple x) (double (double x)))
#void
Scheme>(quadruple 15)
60
```

Now try using the predefined Scheme procedure `display`.

```
Scheme>(display "Hello, world!")
Hello, world!
#void
```

Here `display` had the side-effect of printing `Hello, world!` to the screen, and returned the value `void#`, which was printed.

What you see on the screen may vary in a couple of ways, neither of which is worrisome. Your system may have printed the return value on the same line as the (side-effect) output of `display`, without a linebreak. Since the main use of `display` is for its effect, its return value is undefined, so you may see something other than `#void`, or nothing at all. You might see this:

```
Scheme>(display "Hello, world!")
Hello, world!
"Hello, world"
```

If you do, it means that in your system `display` returns the object you asked it to display. Then Scheme prints out that return value, with double quotes to tell you it's a string object. This shouldn't be too surprising—remember that Scheme prints out the return values of expressions after evaluating them.

Now try `display`ing a number:

```
Scheme>(display 322)
322
#void
```

### 3.1.7 Booleans and Conditionals

In Scheme, falsity is represented by the value *false*, written `#f`. Conceptually, `#f` is a pointer to a special object, the *false* object.

*Predicates* are procedures that return either `#t` or `#f`, and don't have side effects. Calling a predicate is like asking a true/false question—all you care about is a yes or no answer.

Try out the "greater-than" predicate `>`.

```
Scheme>(> 1 2)
#f
```

Here we told Scheme to apply the predicate procedure to 1 and 2; it returned `#f` and Scheme printed that.

The important thing about `#f` is its use in conditionals. If the first subexpression (the *condition*) of an `if` expression returns the value `#f`, the second subexpression is not evaluated, and the third one is; that value is returned as the value of the if expression.

Try just using the literal value `#f` as the first subexpression of an `if`, i.e., the "condition" that controls which branch is taken.

```
Scheme>(if #f 1 2)
2
```

Here the second subexpression was just the literal `2`, so 2 was returned.

Now try it using the predicate `>`

```
Scheme>(if (> 1 2) 1 2)
2
```

This is clearer if we indent it like this, lining up the "then" part (the *consequent*) and the "else" part (the *alternative*) under the condition.

```
Scheme>(if (> 1 2)
           1
           2)
2
```

This is the right way to indent code when writing a Scheme program in an editor, and most Scheme systems will let you indent code this way when using the system interactively—the you can hit `<RETURN>`, and type in extra spaces. Scheme won't try to evaluate the expression until you write the last closing parenthesis and hit `<RETURN>`. This helps you format your code readably even when typing interactively, so that you can see what you're doing.

The false value makes a conditional expression (like an `if`) go one way, and a true value will make it go another. In Scheme, *any value except* `#f` counts as true in conditionals. Try this:

```
Scheme> (if 0 1 0)
```

What result value does Scheme print?

One special value is provided, called the *true* object, written `#t`. There's nothing very special about it, though—it's just a handy value to use when you want to return a true value, making it clear that *all* you're doing is returning a true value.

```
Scheme>(if #t 1 2)
1
```

```
Scheme>(if (> 2 1) 1 2)
1
```

Now let's interactively define the procedure `min`, and then call it:

```
Scheme> (define (min a b)
          (if (< a b)
              a
              b))
#void
Scheme> (min 30 40)
30
```

### 3.1.8 Sequencing

The Scheme system lets you type one expression, then it evaluates it, prints the result, and prompts you for another expression. What if you want to type two or three expressions and have them executed sequentially, i.e., in the written order? You can use a `begin` expression, which just sequences its subexpressions, and returns the value of the last subexpression in the sequence.

First let's define a flag variable, which we'll use to hold a boolean value.

```
Scheme> (define flag #f)
#void
```

Now a sequence to "toggle" (reverse) the value of the flag and return the new value. If the flag holds `#f`, we set it to `#t`, and vice versa.

```
Scheme> (begin (if flag
                   (set! flag #f)
                   (set! flag #t))
              flag)
#t
```

This evaluated the `if` expression, which toggled the flag, and then the expression `flag`, which fetched the value of the variable `flag`, and returned that value.

We can also write a procedure to do this, so that we don't have to write this expression out next time we want to do it. We won't need a `begin` here, because the body of a procedure is automatically treated like a `begin`—the expressions are evaluated in order, and the value of the last one is returned as the return value of the procedure.

```
Scheme> (define (toggle-flag)
           (if flag
               (set! flag #f)
               (set! flag #t))
           flag)
#void
```

Now try using it.

```
Scheme>flag
#t
Scheme>(toggle-flag)
#f
Scheme>flag
#f
Scheme>(toggle-flag)
#t
```

### 3.1.9 Other Flow-of-control Structures

#### 3.1.9.1 Using `cond`

#### 3.1.9.2 Using `and` and `or`

```
==================================================================
This is the end of Hunk B

At this point, you should go back to the previous chapter and
read Hunk C before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk C, which starts at Section 2.2.5 [Comments], page 25.)

### 3.1.10 Making Some Objects (Hunk D)

```
==================================================================
```

```
Hunk D starts here:
==================================================================
```

I've been talking about "objects," but most of the objects we've seen don't have interesting structure.

One of the most important kinds object in Scheme is the *pair*, which you can create with the built-in procedure `cons`. A pair is a simple kind of structured object, like a Pascal record or a C struct. It has two fields, called the *car* and the *cdr*, and you can extract their values with the procedures `car` and `cdr`.

`cons` takes two arguments, which it uses as the initial values of the car and cdr fields of the pair it creates. (`cons` is called that because it `cons`tructs a pair; the name is short because it's a common operation. In Lisp, pairs are called "cons cells" because you make them with `cons`.)

I'll show you some simple examples of playing with pairs, just to show you what they are. Be warned that these are *bad* examples, in that there are usually cleaner ways to do things, which we'll discuss later when we get to lists. (Lists are made of pairs.)

```
Scheme>(cons 1 2)
(1 . 2)
```

What happened here was that the call to `cons` created a pair, and returned (a pointer to) it. Scheme printed out a textual representation of the pair, showing the values of its car and cdr fields.

We didn't do anything with the pair except let Scheme print it, so we've lost it—we didn't save a pointer to it, so we can't refer to it. (The garbage collector will take back its space, so we don't have to worry that we've lost storage space.)

Let's try again, defining (and binding) a variable, and initializing it with the pointer that `cons` returns.

```
Scheme>(define my-pair (cons 1 2))
#void

Scheme>my-pair
(1 . 2)
```

Now try extracting the values of the pair's fields, using `car` and `cdr`. (In Scheme, `(car foo)` is equivalent to C's `foo->car`, dereferencing a pointer to an object and extracting the value of the `car` field. Likewise , `(cdr foo)` is like `foo->cdr`. The operators that access fields of a pair are just procedures.)

```
Scheme>(car my-pair)
1

Scheme>(cdr my-pair)
2
```

We don't need to use any special pointer syntax to dereference the pointer to the pair—*car* and *cdr expect* a pointer, and return the field values of the pair it points to.

`car` and `cdr` only work on pairs. If you try to take the car or cdr of anything else, you'll get a runtime type error.

Try it:

```
Scheme>(car #t)
ERROR: attempt to take the car of a non-pair #t
break>,top
Scheme>
```

The messages you'll see vary from system to system, but the basic idea is the same. We tried to take the `car` of the boolean `#f`, which makes no sense because it has no car field—it doesn't have *any* fields. Scheme told is it didn't work, and gave us a break prompt for sorting it out. Then we just used the `,top` command (or whatever works on your system) to tell Scheme to give up on evaluating that expression and go back to normal interaction.

Scheme also supplies procedures to change the values of a pair's fields, called `set-car!` and `set-cdr!`. They take two arguments, a pair and a value for the field being set.

```
Scheme>(set-car! my-pair 4)
#void

Scheme>my-pair
(4 . 2)

Scheme>(set-cdr! my-pair 5)
#void

Scheme>my-pair
(4 . 5)
```

The value of the variable `my-pair` hasn't actually changed, even though it prints differently. `my-pair` still holds a pointer to the same object, the pair we created with `cons`. What *has* changed is the *contents of* that object. Its fields are like variable bindings, in that they can hold (pointers to) any kind of object, and we've assigned new values to them. (They're *value cells*.)

We can refer to the same object by another name if we just define another variable and initialize it with `my-pair`'s value.

```
Scheme> (define same-pair my-pair)
#void

Scheme>same-pair
(4 . 5)
```

Now suppose we assign a new value to the car of the pair, referring to it via `my-pair`

```
Scheme>(set-car! my-pair 6)
#void

Scheme>my-pair
(6 . 5)

Scheme>same-pair
(6 . 5)
```

Notice that the change is visible through `same-pair` as well as `my-pair`, because we've changed the object that both of them point to.

Now let's make another pair with the same field values.

```
Scheme>(define different-pair (cons 6 5))
different-pair

Scheme>different-pair
(6 . 5)

Scheme>my-pair
(6 . 5)

Scheme>same-pair
(6 . 5)
```

Notice that we have two different pairs, but Scheme prints them out the same way, because it just shows us the *structure* of data structures. We can't tell that they're different just by looking at the printed output. From the printed representation, we can't tell whether or not `my-pair`, `same-pair`, and `different-pair` hold the same values.

Scheme provides a predicate procedure, `eq?`, to tell whether two objects are the exact same object.

```
Scheme>(eq? my-pair same-pair)
#t
Scheme>(eq? my-pair different-pair)
#f
Scheme>(eq? same-pair different-pair)
#f
```

`eq?` tests object *identity*, like pointer comparisons in C (using `==`) or Pascal (using `=`).

It may be confusing, but in programming language terminology, two objects are called *identical* only if they are the very same object, not just two objects that look alike, like "identical" twins. When the government issues "identity" cards, this is the kind of "identity" we're talking about. Two so-called identical twins have different identities, because they're actually different people. A pointer is like a a social security number, because it uniquely identifies a particular individual object.

Scheme also has a test to see whether objects "look the same," that is, have the same structure. It's called `equal?`. We call this a *structural equivalence* test.

```
Scheme>(equal? my-pair same-pair)
#t
Scheme>(equal? my-pair different-pair)
#t
Scheme>(equal? same-pair different-pair)
#t
```

`different-pair` is `equal?` to `my-pair` and `same-pair` because it refers to the same kind of object, and its field values are `equal?`. Notice that that's a recursive definition, which we'll discuss more when we get to lists.

If we didn't have `eq?`, we could still figure out whether two objects were exactly the same object, by changing one and seeing if the other changed, too.

```
Scheme>(set-car! my-pair 4)
#void

Scheme>my-pair
(4 . 5)

Scheme>same-pair
(4 . 5)

Scheme>different-pair
(6 . 5)
```

Now I should warn you about `set-car!` and `set-cdr!`. The reason we put an exclamation point in the name of a procedure that side-effects data is because it's dangerous. If you have two pointers to the same data from different places, i.e., different variable bindings or data structures, it's hard to reason about how changes from one of those places affect things at the other place.

Usually, we like to be able to share data structures, for example, having two data structures that both hold pointers to some third data structure, so the we don't need two copies of it. We don't want to have subtle interactions between different procedures that operate on shared data structures.

You should only use side effects when you have a very good reason to, and make it clear that that's what you're doing. Later examples will show how to program in a style that uses very few side effects, and only where they make sense.

Notice that `cons` is *not* considered a side-effecting operation, because it returns a *new* object that has never been seen before. Somewhere in the implementation of the language, `cons` side-effects memory to initialize it, but you don't see that—from your program's point of view, you're getting a *new* piece of memory that magically has values in place.

```
==================================================================
This is the end of Hunk D.

At this point, you should go back to the previous chapter and
read Hunk E before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk E, which starts at Section 2.2.10 [The Empty List], page 37.)

### 3.1.11 Lists (Hunk F)

```
==================================================================
Hunk F starts here:
==================================================================
```

We usually use pairs in Scheme in a particular, stereotyped way, to build *lists*.

Pairs are really like binary tree nodes—you *can* use the car and cdr fields in the same ways. The normal way of using them treats the car and the cdr differently, however.

The cdr field of a pair is used to hold a pointer to another pair, or a pointer to the empty list, i.e., a null pointer. This lets you string pairs to gether to make linked lists of pairs. The car fields of the pairs hold pointers to any kind of object you want to put in a list.

We can therefore define the term `list` recursively as

- an empty list, i.e., the null pointer object `()`, *or*
- a pair whose `cdr` value is a list.

Think about this, and make sure that you understand why this covers all null-terminated lists strung together by the `cdr`s.

We usually think of lists as holding a sequence of values—we ignore the actual pairs, and think about their `cdr` values.

Because this is how lists are usually used, Scheme has a special way of printing lists. In the earlier examples, I showed that the result of `(cons 1 2)` prints as `(1 . 2)`. You might think that `(cons (cons 1 2) (cons 3 4))` would print as `((1 . 2) . (3 . 4))`, but it doesn't.

The reason is that when Scheme encounters a pair whose `cdr` points to another pair or the empty list, it assumes you want to think of it as a list of pairs strung together by the `cdr`s, and it only shows you the car values.

Try this in your system:

```
Scheme>'()
()
Scheme>(cons 1 '())
(1)
Scheme>(cons 1 (cons 2 '()))
(1 2)
Scheme>(cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Notice that the data structure that prints as `(1 2 3)` is really a binary tree, and we *could* draw it like this:

```
\
 \
  +---+---+
  | * | * |
  +/--+---\
  /        \
 1          +---+---+
            | * | * |
            +/--+---\
            /        \
           2          +---+---+
                      | * | * |
                      +/--+---+
                      /
                     3
```

We generally wouldn't, though, because we think of it as a sequence of numbers, and the pairs are just there to string them together in order. We'd draw it more like this, using the box-and-arrow notation from the previous chapter:

```
     +---+---+     +---+---+    +---+---+
 --->| * | *-+---->| * | *-+--->| * | * |
     +-+-+---+     +-+-+---+    +-+-+---*
       |             |            |
       |             |            |
       1             2            3
```

We've really just rotated the picture 45 degrees, so that "down and to the right" in the tree goes straight right, and looks more like "next" in a linear list.

(The arrow coming in from the left represents the pointer value that was returned, which the read-eval-print loop handed to `write` so that we could see the printed representation of the data structure.)

Drawing things this way lets us show shared structure, if a list overlaps with another list, e.g, if one list joins with the other because some car in each list points at the same object.

Note that a list of this form always ends with a pair whose `cdr` is `()`, (i.e., the empty list, a.k.a. the null pointer).

If we had forgotten that, we might have tried to construct the list this way, with the innermost `cons` just consing two numbers together:

```
Scheme>(cons 1 (cons 2 3))
(1 2 . 3)
```

This is a common beginning mistake. We have constructed an *improper list*—one which is not null-terminated. It doesn't end with `()`.

We could draw the list this way:

```
        +---+---+       +---+---+
   --->| * | *-+------>| * | *-+---->3
        +-+-+---+       +-+-+---+
          |               |
         \|/             \|/
          1               2
```

Notice the dot in `(1 2 . 3)`—that's like the dot in `(2 . 3)`, saying that the cdr of a pair points to `3`, not another pair or `'()`. That is, it's an *improper list*, not just a list of pairs. It doesn't fit the recursive definition of a list, because when we get to the second pair, its cdr isn't a list—it's an integer.

Scheme printed out the first part of the list as though it were a normal *cdr*-linked list, but when it got to the end, it couldn't do that, so it used "dot notation."

You generally shouldn't need to worry about dot notation, because you should use normal lists, not improper list. But if you see an unexpected dot when Scheme prints out a data structure, it's a good guess that you used `cons` and gave it a non-list as its second argument—something besides another pair or `()`.

Scheme provides a handy procedure that creates proper lists, called `list`. `list` can take any number of arguments, and constructs a proper list with those elements in that order.

```
Scheme>(list 1 2 3 4)
(1 2 3 4)
```

We could draw the result like this:

```
    +---+---+      +---+---+      +---+---+      +---+---+
--->| * | *-+----->| * | *-+----->| * | *-+----->| * | * |
    +-+-+---+      +-+-+---+      +-+-+---+      +-+-+---+
      |              |              |              |
     \|/            \|/            \|/            \|/
      1              2              3              4
```

Like any other procedure, `list` can be used with arguments that are procedure calls, such as calls to `list` itself.

```
Scheme>(list (list 1 2) (list 3 4))
((1 2) (3 4))
```

We can draw the result like this:

```
    +---+---+                     +---+---+
--->| * | *-+-------------------->| * | * |
    +-+-+---+                     +-+-+---+
      |                             |
     \|/                           \|/
    +---+---+      +---+---+      +---+---+      +---+---+
    | * | *-+----->| * | * |      | * | *-+----->| * | * |
    +-+-+---+      +-+-+---+      +-+-+---+      +-+-+---+
      |              |              |              |
     \|/            \|/            \|/            \|/
      1              2              3              4
```

While Scheme prints lists in normal list notation when it can, and only uses dot notation when it has to, it can read either one.

We can type in literal lists using the `quote` special form, which just returns a list of the form we typed:

```
Scheme>(quote (1 2 3 4))
(1 2 3 4)
```

Since Scheme can read dot notation, we can do this in an equivalent way, using parentheses around the contents of each pair, and a dot to separate the car and cdr values:

```
Scheme> (quote (1 . (2 . (3 . ( 4 . ())))))
(1 2 3 4)
```

The difference between `list` and `quote` is that `list` is just a procedure, and each time you call it, it creates a new list. The arguments to `list` can be any expressions you like, and their results are what's put in the list.

```
Scheme>(list (double 1) (double 2) (double 3) (double 4))
(2 4 6 8)
```

On the other hand, `quote` is a special form, which only takes one argument, which is *not evaluated at all*—it's just a textual representation of a data structure.

```
Scheme>(quote (double 1))
(double 1)
```

What happened here is that quote just returned a data structure, the list `(double 1)`. It did not try to interpret it as an expression and give its value.

(The first item in the list is the *symbol* `double`. A symbol is just another kind of data object, roughly like a string, which we'll discuss later. It's not the same thing as a variable, even though it prints like a variable name.)

Quoting is so common that Scheme provides a special bit of syntactic sugar to make it easier. In instead of writing out `(quote` before an expression, and a closing parenthesis after, you can just use the special character `'`. Whatever follows should be the textual representation of a data structure, and Scheme constructs that literal data structure.

```
Scheme>'(1 2 3 4)
(1 2 3 4)

Scheme>'((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))

Scheme>'(#f #t)
(#f #t)
```

Later, I'll talk about quoting things besides lists. Quoted lists are enough for now—we'll use them a lot in examples.

```
==================================================================
This is the end of Hunk F.

At this point, you should go back to the previous chapter and
read Hunk G before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk G, which starts at Section 2.4 [Type and Equality Predicates], page 44.)

## 3.2 Using Predicates (Hunk H)

```
==================================================================
Hunk H starts here:
==================================================================
```

Suppose we want to sum a list of numbers.

We can write a procedure `list-sum` to do that, like this:

```
Scheme> (define (list-sum lis)
           (if (null? lis)   ; if empty list?
               0             ; then sum is zero
               (+ (car lis)  ; else it's car plus the
                  (list-sum (cdr lis))))) ; sum of rest of list
#void
```

Try typing in this example, or cutting and pasting it from this file into your running Scheme system. (If you're reading this in a web browser, that should be easy—just cut the text from the browser window, and paste it into your Scheme window at the prompt.) Cutting and pasting is a lot easier than typing in the whole thing!

This procedure accepts one argument, `lis`, which should be a list. It checks to see whether the list is empty, i.e., a null pointer, using the predicate `null?`. If so, it returns `0` as the sum of the elements in the list.

If the list is not empty, the sum of the elements is the sum of the `car` value, plus the sum of the elements in the rest of the list. In that case, `list-sum` takes the car of the list and the `list-sum` of the rest of the list, adds them together, and returns the result.

Try calling this procedure with some lists of numbers, e.g.,

```
Scheme>(list-sum '(1 2 3))
6
Scheme>(list-sum '(4 5 6))
15
Scheme>(list-sum (cons 1 (cons 2 (cons 3 '()))))
6
```

The addition procedure `+` works with floating-point numbers, not just integers, so we can call list-sum with a list of floats as well as integers. (As in most languages, floating point numbers are written with a period to represent the decimal point. Note that there is *no space* between the digits and the decimal point, so that Scheme won't confuse this with dot notation for lists.)

```
Scheme>(list-sum '(1 2.2 3.3))
```

We can modify `list-sum` to print out its argument at each call. Then we can watch the recursion:

```
Scheme> (define (list-sum lis)
           (display "in list-sum, lis is: ")
           (display lis)
           (newline)       ; write a linebreak
           (if (null? lis)    ; if empty list?
               0              ; then sum is zero
               (+ (car lis)   ; else it's car plus the
                  (list-sum (cdr lis))))) ; sum of rest of list
#void
```

### 3.2.1 Using Type Predicates

We can generalize the procedure `tree-sum` to give us the sum of nested lists of integers. The routine `pair-tree-sum` will take a list of integers, or a list of lists of integers, or a mixed list of such lists, and return the sum of all of the integers in the nested lists at any level.

`pair-tree-sum` will handle improper lists as well as proper ones; it's easier that way.

We call this `pair-tree-sum` because we're really using nested lists as a binary tree of integers. (Remember that a Scheme list is just a binary tree, where we're usually think of "down and to the right" as "next" in a linear list. In this case, we're going to go down and to the left as well, to compute the sum of nested lists.

In this case, the sum of a pair-tree is 0 if the pair tree is '(), and if it's a pair, the sum of the whole tree is the sum of its left subtree plus the sum of its right subtree.

Our recursive procedure will have to deal with three cases:

- null lists, e.g., when it's called on the cdr of the last pair in a list
- integers, e.g., when it's called on the car of a list of integers, and
- pairs, e.g., when it's called on the car of a list whose car is another list (i.e., a sublist), or when it's called on the cdr of a list that's not the end of a list.

If we view a list as a binary tree of pairs, the first two cases are the leaves of the tree—either a number, or a null child pointer.

The third case is the recursive case, where we have a subtree to sum.

Our procedure will work for improper lists as well as proper lists, because we'll treat the car and the cdr the same way, as "left child" and "right child" pointers of a tree.

```
Scheme>(define (pair-tree-sum pair-tree)
          (cond ((null? pair-tree)
                 0)
                ((number? pair-tree)
                 pair-tree)
                (else
                 (+ (pair-tree-sum (car pair-tree))
                    (pair-tree-sum (cdr pair-tree))))))
```

Try this out, at make sure you understand why it works.

```
Scheme>(pair-tree-sum '())
0
Scheme>(pair-tree-sum 1)
1
Scheme>(pair-tree-sum '(1 . 2))
3
Scheme>(pair-tree-sum '(1 2))
3
Scheme>(pair-tree-sum '((1 2) 3))
6
Scheme>(pair-tree-sum '((40 . 30) . (20 . 10)))
100
Scheme>(pair-tree-sum '((40 30) (20 10)))
100
```

Add `display` and `newline` expressions at the beginning of `pair-tree-sum`, as we did for `list-sum`, and try it out again. Be sure you understand the output in terms of the recursive call pattern.

### 3.2.2 Using Equality Predicates

Suppose that Scheme didn't provide the predicate `equal?` to do structural comparisons. We could write our own, because we have other type and equality predicates.

Let's write a simplified version of equal that works for lists, including nested lists. We'll consider objects to be `our-equal?` if they are either

- exactly the same objects or equivalent numbers, i.e., they're `eqv?`, or

- if they're both pairs whose cars are `our-equal?` and whose cdrs are also `our-equal?`.

That is, we'll test lists recursively for structural equivalence, "bottoming out" when we hit something that's not a pair. This is pretty much what the standard Scheme predicate `equal?` does, except that it can handle structured data types besides pairs. (For example, it considers two strings with the same character sequence `equal?`, even if they're two different objects.)

```
Scheme>(define (our-equal? a b)
           (cond ((eqv? a b)
                  #t)
                 ((and (pair? a)
                       (pair? b)
                       (our-equal? (car a) (car b))
                       (our-equal? (cdr a) (cdr b)))
                  #t)
                 (else
                  #f)))
```

This procedure checks the easy case first (which is usually a good idea): if two objects are `eqv?`, they're also `our-equal?`.

Otherwise, they're only `our-equal?` if they're both pairs and their cars are equal and their cdrs are equal. Notice the use of `and` here. We first check to see that they're pairs, and then take their cars and cdrs and compare those. If they're not pairs, we won't ever take their cars and cdrs. (If we did, it would be an error, but we rely on the fact that `and` tests things sequentially and stops when one test fails.)

Try it out:

```
Scheme>(our-equal? '() '())
#t
Scheme>(our-equal? 1 1)
#t
Scheme>(our-equal? 1 2)
#f
Scheme>(our-equal? '(1) '(1))
#t
Scheme>(our-equal? '(1) '())
#f
Scheme>(our-equal? '(1 (2)) '(1 (2)))
#t
Scheme>(our-equal? '(((3) 2) 1) '(((3) 2) (1)))
#f
Scheme>(our-equal? '((#f . #t) . (#f . #t))
                   '((#f . #t) . (#f . #t)))
#t
```

```
==================================================================
This is the end of Hunk H

At this point, you should go back to the previous chapter and
read Hunk I before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk I, which starts at Section 2.4.3 [Choosing Equality Predicates], page 47.

## 3.3 Local Variables, `let`, and Lexical Scope (Hunk J)

```
==================================================================
Hunk J starts here:
==================================================================
```

[ to be written ]

```
==================================================================
This is the end of Hunk J

At this point, you should go back to the previous chapter and
read Hunk K before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk K, which starts at Section 2.7 [Procedures], page 58.)

## 3.4 Using First-Class, Higher-Order, and Anonymous Procedures (Hunk L)

```
==================================================================
Hunk L starts here:
==================================================================
```

In this section, we'll play with Scheme's procedures, to illustrate

- *first class procedures*, i.e., normal data objects in the language,
- *higher order procedures*, can take procedures as arguments and return them as values), and
- *anonymous procedures*, which can be created and referred to via pointers, without giving them names)

I'll just briefly demonstrate those ideas for now; later programming examples will show how they're really useful.

### 3.4.1 First-Class Procedures

Scheme procedures are first-class objects in the language; you refer to a procedure in the same way you refer to any other object, via a pointer. A "procedure name" is really just a variable name, and you can do the same things with "procedure" variables as with any other variable. There's really only one kind of variable in Scheme, and it's type is "pointer to anything."

When we "define a procedure" in Scheme, we're really just defining a variable and giving it an initial value that's (a pointer to) a procedure object.

The procedure defining syntax with parentheses around the procedure name (and argument names) is really just syntactic sugar, i.e., a convenient way of writing something that you could do in another way. [ do I use "syntactic sugar" earlier? If so, define earlier. ]

For example,

```
Scheme>(define (double x)
          (+ x x))
#void
```

is *exactly* equivalent to

```
Scheme>(define double
             (lambda (x)
                (+ x x))
#void
```

Try this latter version in your system. Notice that what you're doing is just defining a variable named `double` and initializing it with the result of the second expression, a *lambda* expression.

`lambda` is the real procedure-creating operation. It's a special form, because it lets you define a new procedure rather than calling an existing procedure in the normal way. `lambda` creates a procedure object and returns a pointer to it.

(The predicate `procedure?` can be used to tell if an object is a procedure.)

You can call the `double` procedure created this way in exactly the same way as one created with the sugared procedure-definition syntax.

```
Scheme>(double 3)
6
```

Recall how procedure calls really work. When you call a named procedure, e.g., `(double 3)`, the procedure name is really just a reference to a variable. The first position in the procedure call form is just an expression that's evaluated like any other. In this case, we're using the name `double` as an expression, effectively saying "look up the value of `double`."

Try this

```
Scheme>double
#<procedure>
```

Notice that we didn't put parentheses around `double`, so we're not calling it—we're fetching the value of the variable `double`. What you see on your screen may vary, but it's your system's printed representation of a procedure object. Take a look at it, because you'll want to be able to recognize procedure objects in data structures.

(The printed representation may include the name of the procedure; don't be misled by this. Procedures don't really have names—they're just data objects you can have pointers to, as I'll explain shortly. Your system your system may put a name inside the procedure when you use the procedure definition syntax, but it's just an annotation saying what the procedure's *original* name was—i.e., when it was first defined.)

We can call a procedure in other ways, though—the first subexpression of a procedure call can be any expression we want, as long as it returns a procedure. That expression is evaluated just like the argument expressions—after it and the argument expresssions are evalutated, the resulting procedure is called with those argument values.

```
Scheme>(define list-holding-double (list double))
#void

Scheme>list-holding-double
(#<procedure>)

Scheme>((car list-holding-double) 5)
10
```

What we did here was to create a list holding the procedure formerly known as `double`, and looked at that list. Then we called that procedure by using the expression `(car list-holding-double)` as its "name."

What this shows is that procedures are really *anonymous*, that is, a procedure doesn't have a name in a direct sense. There are just expressions we can refer to it by, if those expressions result in pointers to the procedure.

We can create procedures without normal names at all, by just using *lambda*. Let's create another doubling procedure by just evaluating a `lambda` expression:

```
Scheme>(lambda (x) (+ x x))
#<procedure>
```

The `lambda` expression just created a procedure and returned a pointer to it, and Scheme displayed it however your system does it. We didn't keep a pointer to the procedure, so we can't call it now. The procedure is gone and the garbage collector will clean it up.

We could try again, creating a procedure and keeping a pointer to it in a named variable. More interestingly, we can just hand the pointer to a procedure call, and call it without ever giving it a name.

```
Scheme>((lambda (x) (+ x x)) 6)
12
```

It may not look like it, but this is just a procedure call expression, where the "name" of the procedure is a `lambda` expression to create the procedure we need, and its argument is 6. Note the nesting of parentheses—this is just like `(double 6)`, except that we give the "definition" of the procedure to call, instead of its name.

Later we'll show why using `lambda` directly is often much more convenient than having to name all of our procedures. I'll also explain why `lambda` is the most important special form in Scheme—it is so powerful that most of the special forms can easily be translated into it.

(You might be concerned that creating a procedure and just using it once is very expensive, but it turns out not to be—I'll explain that later, too. For now, don't worry about it.)

### 3.4.2 Higher-Order Procedures

A *higher-order* procedure is one that can take procedures as arguments and/or return them as values. We can use that to write generic procedures that do a basic kind of thing, and take arguments that specialize its behavior.

Here's a simple example.

Scheme provides a procedure `display`, which can write textual representation of a data object on the screen, much like the way the read-eval-print loop displays results of expressions you type in. (This is a very handy procedure for debugging, as well as for programs that interact with users.)

Suppose, though, that you want to display a list of objects, not just one. You want a routine `list-display` to iterate over a list, and display each item in it. The obvious way to write it is to just call `display` from inside your `list-display` routine.

Here's a version like that:

```
Scheme>(define (list-display lis)
          (if (pair? lis)
              (begin (display (car lis))
                     (list-display (cdr lis)))))
```

I've written this procedure recursively, because it's easy to use recursion over lists—usually it's easier than using an iteration construct. This procedure checks to see if what it got was a pair, and if so, it displays the first item, and then calls itself recursively to display the rest of the list. I used a `begin` to sequence the displaying and the recursive call.

It would be cleaner to use `cond`, so here's an equivalent version using `cond`:

```
Scheme>(define (list-display lis)
          (cond ((pair? lis)
                 (display (car lis))
                 (list-display (cdr lis)))))
```

(Notice that this is a one-branch conditional, but we use `cond` instead of `if` because a `cond` branch can be a sequence.)

Now try it out:

```
Scheme>(list-display '(1 2 3))
123#void
```

What happened here is that it displayed each item in the list as it was evaluated, and then Scheme printed out the return value.

(We hadn't specified a return value—the procedure stops when it gets to an the end of the list, and doesn't take the first branch of the one-branch `if` The return value is whatever your Scheme system uses as the result of a one branch `if` when the branch is not taken. It may suppress the printing of an unspecified value, so you may not see it at all.)

This works, but the procedure is not very general. Iterating over lists is very common, so it would be nice to have a more general procedure that iterates over lists, and applies whatever procedure you want.

We can modify our procedure to do this. Instead of taking just a list argument, it can take an argument that's a procedure, and apply that procedure to each element of the list.

We'll call our procedure `list-each`, because it iterates over a list and does whatever you want to each element.

```
Scheme>(define (list-each proc lis)
          (if (pair? lis)
              (begin (proc (car lis))
                     (list-each proc (cdr lis)))))
```

The only change we made was to add an argument `proc`, to accept (a pointer to) a procedure, and to change the call to `display` into a call to `proc`.

Now we can call this general procedure with the argument `display`, to tell it to `display` each thing in the list.

```
Scheme>(list-each display '(1 2 3))
123#void
```

But maybe this isn't what we want. We might want to print each item, and then a newline (go to the next line), to spread things out vertically. We could write a procedure `display-with-newline` to do that, but it's easier just to use a `lambda` expression to create the procedure we need.

Try this:

```
Scheme>(list-each (lambda (x)
                     (display x)
                     (newline))
                  '(1 2 3))
1
2
3
#void
```

(Scheme has a standard procedure similar to our `list-each`, but more general, called `for-each`.)

Rewrite `list-each` to use a one-branch `cond` and try it out.

```
==================================================================
This is the end of Hunk L

At this point, you should go back to the previous chapter and
read Hunk M before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk M, which starts at Section 2.7.4 [lambda and Lexical Scope], page 62.)

## 3.5 Interactively Changing a Program (Hunk N)

```
==================================================================
Hunk N starts here:
==================================================================
```

### 3.5.1 Replacing Procedure Values

Earlier we showed how to replace normal data values in variable bindings, using the side-effecting special form `set!`.

We can also change procedure values. One way of doing this is just to change the value of the procedure variable. (Remember that a named procedure is really just a procedure object that happens to have a pointer to it stored in that variable.)

Just as we changed the value of the variable `myvar` using `set!`, we can change the value of the procedure variable `quadruple`. Try this:

```
Scheme>(quadruple 3)
12
Scheme>(set! quadruple double)
#[procedure]
Scheme>(quadruple 3)
6
```

What happened here is that when we evaluated the expression `(set! quadruple double)` it just did the usual thing `set!` does when both of its arguments are variables—it computed the value of the expression on the right, in this case by fetching the value from the binding of `double`, and stored it into the (binding of) the variable on the left. In this case, the value of `double` is (a pointer) to a procedure—the one that we created when we `define`'d `double`. This pointer was copied into `quadruple`, so that it now contains a pointer to the very same procedure.

Calling `quadruple` now has the same effect as calling `double`, because either way, a pointer is fetched from the variable, and whatever it points to is called.

Note that while this illustrates how Scheme works, and we'll show why it's handy later, it's not usually a great idea to go around changing the values of procedure variables by side-effecting them with `set!`.

Usually, once a program has been developed, you don't want to clobber named procedures, because it makes the code hard to understand—you don't want your finished program to go around changing the meaning of procedure names as it runs. (You normally want to be able to look at your program and see the definitions, and not have to worry that some other part of the program may change the procedures at odd moments.)

During interactive development of a program, however, it's often very convenient to be able to change a procedure's behavior at will. (We're not really modifying a procedure, though—we're changing a variable binding's value to affect which procedure is called. We don't have to actually modify any procedure objects, because we can replace a pointer to one procedure with a pointer to another.)

Usually you'll want to do this by *redefining* the procedure with another `define` expression.

For example, suppose we want to restore the old behavior of `quadruple`, which we foolishly clobbered above. We can simply `define` it again, the old way:

```
Scheme>(define (quadruple x) (double (double x)))
quadruple
```

In a finished program, you generally shouldn't have multiple definitions of the same thing—a `define` form should define something that doesn't change during program execution. If you want to change the state of a binding, use `set!` to make it clear that's what's going on, and put a comment at the definition of the variable warning that it is likely to be changed at runtime.

Most interactive Scheme systems let you `define` the same variables multiple times, though, so that you can change things during program development. (Note that we're talking about redefining the same program variable here, not defining different variables with the same name in different scopes.)

### 3.5.2 Loading Code from a File

When you're actually developing a program, you often want to save the text in a file, rather than just typing it in and losing it when you exit the Scheme system.

The simplest way of doing this is to use an editor in one window and Scheme in another. From the editor, save your program text into a file, and then load it into Scheme with the `load` procedure. `load` takes a string as an argument, which is the name of the file to load, and reads it in just as though you had typed it in by hand, at the prompt. (A string literal is written with double quotes around it; there'll be more about strings more later.)

Type the following text into your editor and save it into a file named `triple.scm`.

```
(define (triple x)
   (+ x (+ x x)))
```

Now, at the Scheme prompt, load the file and call the procedure:

```
Scheme>(load "triple.scm")
loading...triple...done
Scheme>(triple 3)
9
```

(Notice that in the above example, there's no connection between the string we used to name the file, `"triple.scm"`, and the name of the procedure, `triple`. We just chose to call the file `"triple.scm"` to remind us what's in it.)

Usually, when you're developing a program, you should put only a few definitions in a file—maybe just one. This lets you change small parts of your program, saved the changed file, and reload the file to change the definitions in your running Scheme system.

Good editors also have packages that allow you to run Scheme and use an editor command to send the contents of a file (or a selected region of a file) to Scheme, as though you'd typed it in. (Emacs has excellent facilities for this.)

If you're using a graphical user interface, you may be able to simply cut text from your editor, and paste it into the window you have Scheme running in, so that it appears to Scheme as though you'd just typed it in.

Be careful about reloading definitions. When you load a file, the Scheme system will reuse the same top-level bindings, and reinitialize them. In general, new objects will be constructed, even if the textual definitions haven't changed.

For example, suppose we have the following code in a file, which we've already loaded once:

```
(define my-list (list 1 2))

(define my-other-list (cdr my-list))
```

If we reload this file, all three definitions will be processed again. A new list will be constructed and the existing binding of `my-list` will be updated to point at the new list.

Likewise, the existing binding of `my-other-list` will be updated with the `cdr` of that new list. Each time we reload the file, we'll recreate the intended data structure, including the sharing relationship between the two lists.

But now consider what happens if this code is spread across two files, with the definition of `my-other-list` in a different file, which we don't reload. If we just reload the first definition, then the binding `my-other-list` will still refer to the `cdr` of the *old* list, not the new one. If your code depends on the two lists sharing structure, it not behave as expected, because the two variables' bindings will refer to distinct lists.

Procedures can cause the same sorts of problems. If you have a pointer to a procedure in a data structure, and then you redefine the procedure by modifying the definition and reloading it, a *new* procedure object will be created, but the old data structure will still hold a pointer to the old procedure object.

In general, you should be careful to recreate any data structures holding procedures if you redefine those procedures. This is usually easy, if you reload the code that creates the data structures, after reloading the new definitions of the procedures.

Notice that this is *not* necessary if you just call top-level procedures (or look up variable values) in the usual way. For example, given our earlier definitions of *double* and `quadruple`, changing `double` affects `quadruple` immediately. Every time we call `quadruple`, it fetches the *current* value of the binding of `double`, which ensures that it sees the most recent version. We can reload the code for `double`, without reloading the code for `quadruple`.

### 3.5.3 Loading and Running Whole Programs

[ to be written ]

## 3.6 Some Other Useful Data Types

Scheme has several important kinds of data objects that are useful in programming in general, and particularly for writing an interpreter, as we'll do in the next chapter. These include character strings, symbols, and lists.

Scheme has two data types that represent sequences of characters, called *strings* and *symbols*. Strings are pretty much like character strings in most programming languages—they represent a sequence of text characters. Symbols are sort of like strings, but have a very special property—there's only one symbol object with any particular sequence of characters.

Symbols have a special role in the implementation of Scheme, because they're part of the normal representation of source code; symbols are used to represent names of variables, procedures, special forms, and macros. They're really just a kind of data object, though—you can use them in your programs, whether or not you want to represent code.

Lists are used in interpreters and compilers to represent compound expressions in the source code; nested expressions are generally represented by nested lists.

More generally, there's a category of Scheme data structures called *s-expressions*, which consist of basic types including symbols, strings, numbers, booleans, and characters, and list of those simple types, or lists of such lists.

"S-expression" is short for "symbolic expression," but it's something of a misnomer. An *expression* is really a piece of a program. An "s-expression " is just a data structure, which may or may not represent an expression in a programming language, although interpreters and compilers often happen to use them that way.

### 3.6.1 Strings

Character strings in Scheme are written between double quotes. For example, suppose we want an object that represents the text "Hello world!" We can just write that in a program, in between double quotes: `"Hello, world!"`.

You can use a string as an expression—the value of a string is the string itself, just as the value of an integer is the integer itself. Like numeric literals and booleans, strings are "self-evaluating," which just means that if you have an expression in your program that consists of just a string, Scheme assumes you mean the value to be literally that string. There's nothing deep about this—it just turns out to be handy, because it makes it easy to use strings as literals.

Try typing the string `"Hello, world."` at the Scheme prompt.

```
Scheme>"Hello, world!"
"Hello, world!"
```

What happened here is that Scheme recognized the sequence of characters between double quotes as a string, and constructed a Scheme string object with that sequence of characters. It then evaluated that object as an expression. Recognizing it as a string, it simply returned the string object as the value of the expression. Then, as it always does, Scheme printed out the value of the expression, namely the string object. The standard printed representation of a string object includes the double quotes, so that you know it's a string object.

If you want to print out a string, but without the double quotes, you can use the standard procedure `display`. If you pass `display` a string, it just prints out the characters in the string, without any double quotes. `display` is useful in programs that print information out for normal users. Another useful procedure is `newline`, which prints a newline character, ending a line and starting a new one.

Try typing a `(display "Hello, world!") (newline)` at the Scheme prompt. What you get may look like this:

```
Scheme>(display "Hello, world!") (newline)
Hello, world!
#void
```

You might see something slightly different on your screen, depending on the return value of `newline`, which is unspecified in the Scheme standard.

If you type in an expression using a string literal like `"foo"` at the Scheme prompt, Scheme may construct a new string object with that character sequence each time.

Try this:

```
Scheme>(define foo1 "foo")
#void
Scheme>(define foo2 "foo")
#void
Scheme>foo1
"foo"
Scheme>foo2
"foo"
Scheme>(eq? foo1 foo2)
#f
Scheme>(equal? foo1 foo2)
#t
```

For each of the `define` forms, Scheme has constructed a string with the character sequence `f-o-o`, and saved it in a new variable binding. When we ask the value of each variable, Scheme prints out the usual text representation of the string. The printed representations are the same, since each string has the same structure, but they're two different objects—when we ask if they're `eq?`, i.e., the very same object, the answer is no (`#f`).

It's possible that in your system the `eq?` comparison will return `#t`, because Scheme implementations are allowed to use pointers to the same string if you type in two strings with the same character sequence. For that reason, you should be careful not to depend on whether Scheme strings are `eq?`; you should only distinguish whether they're `equal?`. You can also use the predicate `string-equal?` if you know the arguments are supposed to be strings. This has the advantage of signaling an error if the arguments are of unexpected type.

Strings can be used as one-dimensional arrays (vectors) of characters. There are procedures for accessing their elements by an integer index, extracting substrings given two indices, and so on.

### 3.6.2 Symbols

Symbols are like strings, in that they have a character sequence. Symbols are different, however, in that *only one* symbol object can have any given character sequence. The character sequence is called the symbol's *print name*. A print name is *not* the same thing as a variable name, however—it's just the character sequence that identifies a particular unique symbol.

Unlike strings, booleans, and numbers, symbols are *not* self-evaluating. To refer to a literal symbol, you have to *quote* it. Since print names of symbols look just like variable names, you have to tell Scheme which you mean.

If we type in the character sequence `f o o` *without* double quotes around it, Scheme assumes we mean to refer to a variable named `foo`.

In interpreters and compilers, symbol objects are often used as variable names, and Scheme treats them specially. If we just type in a character string that's a symbol print name, and hit return, Scheme assumes that we are asking for the value of the binding of the variable with that name—if there is one.

```
Scheme>(define foo 10)
#void

Scheme>foo
10
```

If we quote the symbol name with the single quote character, Scheme interprets that as meaning we want the *symbol object* `foo`.

```
Scheme>'foo
foo
```

Since we've already defined (and bound) the variables `foo1` and `foo2`, we can ask Scheme to look up their values.

```
Scheme>foo1
"foo"
Scheme>foo2
"foo"
```

Here we've typed in the names that we gave to variables earlier, and Scheme looked up the values of the variables.

As we've seen before, this doesn't work if there isn't a bound variable by that name. Symbols can be used as variable names, if you define the variable, but by default a symbol is just an object with a particular print name that identifies it.

If we want to refer to the *symbol object* `foo`, rather than using foo as a variable name, we can *quote* it, using the special quote character `'`. This tells Scheme *not* to evaluate the following expression, but to treat it as literal data.

```
Scheme> 'foo
foo
```

The first time you type in a symbol name, Scheme constructs a symbol object with that character sequence, and puts it in a special table. If you later type in a symbol name with the same character sequence, Scheme notices that it's the same sequence. Instead of constructing a new object, as it would for a string, it just finds the old one in the table, and uses that—it gives you a pointer to the same object, instead of a pointer to a new one.

Try this:

```
Scheme>(define bar1 'bar)
#void
Scheme>(define bar2 'bar)
#void
Scheme>(eq? bar1 bar2)
#t
```

Here, when we typed in the first definition, Scheme created a symbol object with the character sequence `b a r`, and added it to its table of existing symbols, as well as putting a pointer to it in the new variable binding `bar1`. When we typed in the second definition, Scheme noticed that there was already a symbol object named `bar`, and put a pointer to that same object in `bar2` as well.

When we asked Scheme if the values of `bar1` and `bar2` referred to the same object, the answer was yes (`#t`)—they both referred to the unique symbol `bar`; there is only one symbol by that name.

The big advantage of symbols over strings is that comparing them is very fast. If you want to know if two strings have the same character sequence, you can use `equal?`, which will compare their characters until it either finds a mismatch or reaches the ends of both strings.

With symbols, you can use `equal?`, but you can get the same results using `eq?`, which is faster. Recall that `eq?` just compares the pointers to two objects, to see if they're actually the same object. For symbols, this works to compare the print names, too, because two symbols can have the same name only if they're the same object. You don't have to worry about symbols being `equal?` but not `eq?`.

This makes symbols good for use as keys in data structures. For example, you can zip through a list looking for a symbol, using `eq?`, and all it has to do is compare pointers, not character sequences.

Another advantage of symbols is that only one copy of its character sequence is actually stored, and all occurrences of the same symbol are represented as pointers to the same object. Each additional occurrence of symbol thus only costs storage for a pointer.

If you're doing text processing in Scheme, e.g., writing a word processor, you probably want to use strings, not symbols. Strings support more operations that make it convenient to concatenate them, modify them, etc.

Symbols are mainly used as key values in data structures, which happen to have a convenient human-readable printed representation.

If you need to convert between strings and symbols, you can use `string->symbol` and `symbol->string`. `string->symbol` takes a string and returns the unique symbol with that print name, if there is one. (If there's not, and the string is a legal symbol print name, it creates one and returns it.) `symbol->string` takes a symbol and returns a string representing its print name. (There is no guarantee as to whether it always returns the same string object for a given symbol, or a copy with the same sequence of characters.)

#### 3.6.2.1 A Note on Identifiers

When you type in a string, e.g., `"This here is a string, you know."`, you can type in pretty much whatever you want, as long as it's between double quotes and doesn't have double quotes or nonprinting characters in the middle. (You can have strings with double quotes in them, but you have to use a special escape sequence trick.)

When you type in a symbol, on the other hand, you have to be a little more careful—some character sequences count as symbol names, but others don't. For example, the character sequence `1 2 3` doesn't count as a symbol `123`, because it's a number. Character sequences with spaces, parentheses, and single quotes in them are also a no-no, because those characters have special meaning when reading and writing the printed representations of Scheme data structures.

A symbol name has to start with an "extended alphabatic" character—that a letter or any of a fairly large set of printing characters, followed by a string of other extended alphabetic characters or digits. (The extended alphabetic characters are `a-z`, `A-Z`, and these: `+ - . * / < = > ! ? : $ % _ & ~ ^`.)

For example, the following are all symbols:

```
x
thursdays-total*3
am_is_are_was_were_be_being_been
able-was-I-ere-I-saw-elba
floppy_drive-3.5
fourscore-and-7-years-ago
x-15+three-times-thirty-seven
=1
lhs=>rhs
x+/-3%
```

There is a slight restriction that you can't use a symbol name that starts with a character that could begin a literal number. This includes not only digits, but `+`, `-`, `.` and `#`. A special exception to this is that `+`, and `-`, by themselves, are symbols, and so is `...` (the ellipsis identifier used in macros).

Scheme identifiers (variable names and special form names and keywords) have almost the same restrictions as Scheme symbol object character sequences, and it's no coincidence. Most implementations of Scheme happen to be written in Scheme, and symbol objects are used in the interpreter or compiler to represent variable names.

Don't read too much into this, however: it's easy to write a Scheme interpreter or compiler in Scheme, and that *is* why the rules for symbol names are the same as the rules for variable names, but symbols and variables are very, very different things. A symbol is just a data object, like a string, that has the special property of being unique. You can use symbols like any other data object, as part of any data structure.

It just happens that interpreters and compilers generally use symbol objects to represent the names of variables and whatnot, so it's *convenient* that the rules for symbol object names are the same as the rules for identifiers in the language—but there is no other connection.

Symbols are *not* necessarily variable names, they're just a kind of data object (like strings) that happen to get used that way, by *some* programs (interpreters and compilers). *Your* programs can use them any old way you choose. (Sorry to be repetitive on this point, but confusing symbols and variables is one of the most common and avoidable problems in learning Scheme. It's worse in Lisp, where symbols and variables do have a deep connection, but not an obvious one.)

### 3.6.3 Lists Again

Suppose we want to make a list of symbols whose print names are the English words for the first five integers. We could do it using quoting, of course, like this:

```
Scheme>(define firstfive '(one two three four five))
#void
Scheme>firstfive
(one two three four five)
```

We don't have to quote each symbol individually. Within a `quote` expression, everything is assumed to be literal data, not expressions to evaluate.

We could also do it by calling `list` to construct the list, and handing it each of the five symbols as literals. To do that, we have to quote them, so that Scheme won't think we're referring to variables named `one`, `two`, etc.

```
Scheme>(define firstfive (list 'one 'two 'three 'four 'five))
#void
Scheme>firstfive
(one two three four five)
```

Since `list` is a procedure, its argument expressions are evaluated. We use a `quote` around each expression, so that it will return a pointer to the appropriate symbol, rather than the value of the variable by the same name.

This works whether or not there *is* a variable by that name, because names of symbols and names of variables are completely different things.

For example, even after evaluating the above expressions, attempting to evaluate the expression `four` will be an error, unless we've defined a variable named `four`. The existence of a symbol with a given print name doesn't say anything about the existence of a variable with that name.

### 3.6.3.1 Heterogeneous Lists

Since Scheme is dynamically typed, we can put any kind of object in a list. So far, we've made a list of integers and a list of symbols. We can also make a list of different kinds of things, such as a list of integers, symbols, and lists.

```
Scheme>(define mixed5 '(one 2 (three and a) "four" 5))
#void
Scheme>mixed5
(one 2 (three and a) "four" 5)
```

Here we've constructed a mixed list whose first element is a symbol, the second is an integer, the third is a list of symbols, the fourth is a string, and the fifth is another integer. (The technical term for a mixed list is a "heterogeneous list.")

We can draw it like this:

```
       +------+
mixed5 |   +--+-->+---+---+  +---+---+  +---+---+  +---+---+  +---+---+
       +------+   | + | +-+->| + | +-+->| + | +-+->| + | +-+->| + | * |
                  +-+-+---+  +-+-+---+  +-+-+---+  +-+-+---+  +-+-+---+
                    |          |          |          |          |
                   \|/        \|/         |         \|/        \|/
                   one         2          |        "four"       5
                                          |
                                         \|/
                                        +---+---+  +---+---+  +---+---+
                                        | + | +-+->| + | +-+->| + | * |
                                        +-+-+---+  +-+-+---+  +-+-+---+
                                          |          |          |
                                         \|/        \|/        \|/
                                        three       and         a
```

Notice that we draw the symbols (`one`, `three`, `and`, and `a`) as simple sequences of characters. This is just a drawing convention. They're really objects, like pairs are. We draw strings similarly, but with double quotes around them. Don't be fooled—these are objects on the heap, too. We just draw them this way to keep the picture from getting cluttered up.

#### 3.6.3.2 Operations on Lists

We've already seen two list-processing procedures that you'll use a lot, `car` and `cdr`. `car` takes a pointer to a pair, and extracts the value of its first (`car`) field. `cdr` takes a pointer to a pair and returns the value of its second (`cdr`) field.

(It might have been better if `car` had been called `first` and `cdr` had been called `rest`, since that's more suggestive of how they're used: a pointer to the first item in a list, and a pointer to the pair that heads the rest of the list.)

Given our list stored in `mixed5`, we can extract parts of the list using `car` and `cdr`.

```
Scheme>(car mixed5)
one
Scheme>(cdr mixed5)
(2 (three and a) "four" five)
```

By using `car` and `cdr` multiple times, we can extract things beyond the first element. For example, taking the `cdr` of the `cdr` of a list skips the first two elements, and returns the rest:

```
Scheme>(cdr (cdr mixed5))
((three and a) "four" 5)
```

Taking the car of that list (that is, the `car` of the `cdr` of the `cdr`) returns the first item in that list:

```
Scheme>(car (cdr (cdr mixed5)))
(three and a)
```

We can keep doing this, for example taking the second element of that sublist by taking the car of its cdr.

```
Scheme>(car (cdr (car (cdr (cdr mixed5)))))
and
```

This starts to get tedious and confusing—too many nestings of procedures that do too little at each step—so Scheme provides a handful of procedures that do two list operations at a whack. The two most important ones are `cadr` and `cddr`.

`cadr` takes the `car` of the `cdr`, which gives you the second item in the list. `cddr` takes the `cdr` of the `cdr`, skipping the first two pairs in a list and returning the rest of the list.

This lets us do the same thing we did above, a little more concisely and readably:

```
Scheme>(cadr (car (cddr mixed5)))
and
```

With a little practice, it's not hard to read a few nested expressions like this. In this example, taking the `cddr` of `mixed5` skips down the list two places, giving us the list that starts with the sublist we want. Then taking the `car` of that gives us the sublist itself off the front of that list, and taking the `cadr` of that gives us the second item in the sublist.

Of course, even if Scheme didn't provide `cadr` and `cdr`, you could write them yourself in terms of `car` and `cdr`:

```
(define (cadr x)
   (car (cdr x)))

(define (cddr x)
   (cdr (cdr x)))
```

Scheme actually provides predefined list operations for all combinations of up to four `car`'s and `cdr`'s. For example, `cadadr` takes the `cadr` of the `cadr`. (The naming scheme is that the pattern of `a`'s and `d`'s reflects the equivalent nesting of calls to `car` and `cdr`.)

You probably won't want to bother with most of those, because the names aren't very intuitive. Two procedures that are worth knowing are `list-ref` and `list-tail`.

`(list-ref` *list n*`)` extracts the *n*th element of a list `list`, which is equivalent to *n-1* applications of `cdr` followed by `car`. For example, `(list-ref '(a b c d e) 3)` is equivalent to `(car (cdr (cdr '(a b c d e))))`, and returns `d`.)

In effect, you can index into a list as though it were an array, using `list-ref`. (Of course, the access time for an element of a list is linear in the index of the element. If you need constant-time access, you can use vectors, i.e., one-dimensional arrays.) Notice that the numbering is zero-based, which is why `(list-ref lis 3)` returns the *fourth* element of a `lis`. This is consistent with the indexing of vectors, which are also zero-based, as well as reflecting the number of `cdr` operations.

`(list-tail` *list n*`)` skips the first *n* elements of a list and returns a pointer to the rest, which is equivalent to repeated applications of `cdr`. (This is standard R4RS Scheme, but not IEEE Scheme. If your Scheme doesn't provide `list-tail`, you can easily write your own.)

These two procedures can make it much clearer what you're doing when you extract elements from nested lists. Suppose that we have a list `foo`, which is a triply-nested list—a list of lists of lists, and we want to extract the second element of the bottom-level list that is the third element of the middle-level list that is the fourth element of the outermost list.

We could write `(car (cdr (car (cdr (cdr (car (cdr (cdr (cdr foo)))))))))`, but that's pretty hard to read. If we use `cadr`, `caddr`, and `cadddr`, we can make it somewhat more readable by using one function call at each level of structure: `(cadr (caddr (cadddr foo)))`. But it's still clearer to write `(list-ref (list-ref (list-ref foo 4) 3) 2)`

or (indented)

```
(list-ref (list-ref (list-ref foo 4)
                    3)
          2)
```

`list-ref` and `list-tail` are much more convenient than things like `caddr` when the indexes into a list vary at run time. For example, we might use an index variable `i` (or some other expression that returns an integer) to pick out the *i*th member of a list: `(list-ref foo i)`. Writing this with `car` and `cdr` would require writing a loop or recursion to perform *n-1* `cdr`'s and a `car`.

```
==================================================================
This is the end of Hunk N

At this point, you should go back to the previous chapter and
read Hunk O before returning here and continuing this tutorial.
==================================================================
```

(Go BACK to read Hunk O, which starts at Section 2.9 [Tail Recursion], page 79.)

## 3.7 Basic Programming Examples (Hunk P)

[ From here on, the text is not structured as a type-along tutorial interleaved with Chapter 2. However, it's a good idea to experiment with the examples interactively in a running Scheme system . ]

In this section, I'll give a few simple examples of Scheme programming, mostly using recursion to manipulate lists without side effects. (Later, I'll revisit some of these examples, and show how to implement them more efficiently, using tail recursion, but still without side effects.)

I'll show how to implement simple versions of some standard Scheme procedures; this may help you understand what those procedures do, and how to use them. (Later, I'll return to some of these examples and show how to implement more general versions.) I'll also give some examples that aren't standard Scheme procedures, but illustrate common idioms.

Some of these examples use higher-order procedures—procedures which operate on procedures—and toward the end of the section, I'll discuss *currying*, a technique for creating specialized versions of procedures in a particular context.

You should get used to thinking recursively, and avoiding side effects most of the time. It's often easier to write things recursively than using normal loops and side effects.

### 3.7.1 An Error Signaling Routine

It's often useful to put error-checking code in your procedures, to make sure that their arguments satisfy whatever preconditions they need to operate correctly.

In a dynamically-typed language, this is often good for making sure that you detect errors where pass values to a procedure that can't handle arguments of those types. Usually when you do that, you'll find out soon enough, because you'll perform an illegal operation (like taking the `car` of a number), and Scheme will detect the error and tell you.

Scheme doesn't yet have a standard error signaling routine, but we can make a simple portable one.

[ In fact, if you don't have one, you'll get an error signaled anyway, in the form of an unbound variable exception when you try to call `error` . . .... ]

[ We should discuss portable and system-specific error-handling in class, ]

### 3.7.2 `length`

`length` is the standard Scheme procedure that returns the length of a list. It only counts the elements along the spine of the list (down the `cdr`'s).

It's easy to do this using recursion. The length of a list is 0 if the list is empty, and otherwise it's 1 plus the length of the rest of the list. Here's the easiest way to define `length`:

```
(define (length lis)
   (if (null? lis)
       0
       (+ 1 (length (cdr lis)))))
```

Note that in this example, I've used `lis` as the name of a list argument, rather than `list`. That's because there's a standard Scheme procedure named `list`, which will be shadowed by any local variable with the same name. (This is because of Scheme's unified namespace; `list` seems to be the only identifier for which this is commonly a problem.)

The above definition of `length` is not tail recursive—after calling itself, there must be a return so that `1` can be added to the value and returned.

Later we'll show a more efficient, tail-recursive version of `length`, and a more general procedure called `reduce` that can be used to construct a variety of procedures whose basic algorithm is similar.

### 3.7.3 Copying Lists

There are two common senses of copying, *shallow* copying, and *deep* copying. A shallow copy makes a copy of one object, and the copy has pointers to the same objects that the original did.

A *deep* copy copies not only the top-level objects in a data structure, but the ones below that, and so on recursively, so that a whole new data structure is created.

For lists, which are made up of more than one object, it is often useful to copy the spine of the list, i.e., doing a deep copy along the `cdr`'s only. We typically think of a list as being like a special kind of object, even though it's really a sequence of pair objects. It's therefore natural to copy "just the list."

If we just want to do a shallow copy, we can define `pair-copy` to copy a pair, without copying anything else.

In these examples, I'll assume we only want to copy list structure—that is a connected set of pairs. Whenever we come to something that's not a pair, we stop copying and the copy shares structure with the original. (These aren't standard Scheme procedures.)

Here's a truly shallow copy, just copying a single pair:

```
(define (pair-copy pr)
   (cons (car pr) (cdr pr)))
```

If we want to do a deep copy, we can use recursion to copy `car` or `cdr` values that are also pairs. The following code for `lists-deep-copy` assumes that the structure to be copied is a tree of pairs. (If there is any shared structure, it will be copied each time it is reached. If there's a directed cycle, `deep-copy` will loop infinitely.)

```
(define (lists-deep-copy thing)
   (if (pair? thing)
       (cons (lists-deep-copy (car thing))
             (lists-deep-copy (cdr thing)))
       thing))
```

To copy the spine of a list, we can use this

```
(define (list-copy lis)
   (cond ((pair? lis)
          (cons (car lis)
                (list-copy (cdr lis))))
         ((null? lis)
          '())))
```

[ need to discuss error checking, tastefulness...]

### 3.7.4 `append` and `reverse`

Two handy operations on lists are `append` and `reverse`; both are standard Scheme procedures.

`append` takes any number of lists as arguments, and returns a list with all of their elements. `reverse` takes a list and returns a new list, with the same elements but in the opposite order.

Note that like most Scheme procedures, neither of these procedures is destructive—each creates a new list without modifying its argument.

#### 3.7.4.1 append

Here's a simple two-argument version of `append`:

```
(define (append2 lis1 lis2)
  (if (null? lis1)
      lis2
      (cons (car lis1)
            (append2 (cdr lis1) lis2))))
```

Note that `append2` copies its first list argument, but the result simply shares a pointer to the last list argument—the last list is not copied, so the result shares structure with that list. This is also true of the standard Scheme function `append`, which can take any number of lists as arguments. The first $n$-1 lists are copied, but the last is shared.

Suppose we have defined two lists, `foo` and `bar`, like this:

```
(define foo '(x y z))
(define bar '(a b))
(define baz (append bar foo))
```

The result will be that `baz` shares structure with `foo`, but not with `bar`. Changes to the list via `foo` will also be visible via `baz`.

```
               +----------------------------------------+
               |                                        |
              \|/                                       |
    +---+    +---+---+     +---+---+      +---+---+     |
foo | *-+--->| * | *-+---->| * | *-+----->| * | * |     |
    +---+    +-+-+---+     +-+-+---+      +-+-+---+     |
               |             |              |           |
              \|/           \|/            \|/          |
               x             y              z           |
                                                        |
                                                        |
    +---+    +---+---+     +---+---+                    |
bar | *-+--->| * | *-+---->| * | * |                    |
    +---+    +-+-+---+     +-+-+---+                    |
               |             |                          |
              \|/           \|/                         |
               a             b                          |
              /|\           /|\                         |
               |             |                          |
    +---+    +---+---+     +---+---+                    |
baz | *-+--->| * | *-+---->| * | *-+--------------------+
    +---+    +---+---+     +---+---+
```

[ give code for full-blown append, using dot notation for variable arity arg list? Do it later, fwd ref from here? ]

### 3.7.4.2 `reverse`

`reverse` returns a reversed copy of a list.

There's an easy (but slow) way to define `reverse` in terms of `append`. We just take the first element off the list, reverse the rest of the list, and append the first element to the end.

```
(define (reverse lis)
   (if (null? lis)
       '()
       (append (reverse (cdr lis))
               (list (car lis)))))
```

Think about how this works. `reverse` recurses down the list, calling itself on the `cdr` of the list at each recursive step, until the recursion stops at the end of the list. (This last call returns the empty list, which is the reverse of the empty list.) At each step, we use `car` to peel off one element of the list, and hold onto it until the recursive call returns.

The reversed lists are handed back up through the returns, with the cars being slapped on the rear of the list at each return step. We end up constructing the new list back-to-front on the way up from the recursion.

There are two problems coding `reverse` this way, and later I'll show better versions. (They'll still be recursive, and won't use loops or assignment.)

The first problem is that each call to `append` takes time proportional to the length of the list it's given. (Remember that append has to copy all of the elements in the first list it's given.) We have to copy the "rest" of the list (using `append` starting at each pair in the list. On average, we copy half the list at a given recursive step, so since we do this for every pair in the list, we have an n-squared algorithm.

Another problem is that we're doing things on the way back up from recursion, which turns out to be more expensive than doing things on the way down. As I'll explain in a later chapter, Scheme can do recursion very efficiently if everything is done in a forward direction—it can optimize away all but one of the returns. (Luckily, this is easy to do.)

### 3.7.5 `map` and `for-each`

`map` and `for-each` are used to apply a procedure to elements of lists. They're good for coding repetitive operations over sets of objects.

#### 3.7.5.1 `map`

`map` takes a procedure and applies it to the elements of a list (or corresponding elements of a set of lists), returning a list of results.

For example, if we want to double the elements of a list, we can use `map` and the `double` procedure we defined earlier:

```
Scheme>(map double '(1 2 3))
(2 4 6)
```

If the procedure we're calling takes more than one argument, we can pass two lists of arguments to `map`. For example, if we want to add corresponding elements of two lists, and get back a corresponding list of their sums, we can do this:

```
Scheme>(map + '(1 2 3) '(4 5 6))
(5 7 9)
```

Right now, we'll just write a simplified version of `map`, which takes one list of values and applies a one-argument procedure to them.

```
(define (map proc lis)
   (cond ((null? lis)
          '())
         ((pair? lis)
          (cons (proc (car lis))
                (map proc (cdr lis))))))
```

Notice that `map` may construct a list of results front-to-back, or back-to-front, depending on the order of the evaluation of the arguments to `cons`. That is, it may apply the `map`ped procedure on the way down during recursion, or on the way back up. (This is allowed by the Scheme standard—the order of the results in the resulting list corresponds to the ordering of the argument list(s), but the dynamic order of applications is not specified.)

### 3.7.5.2 `for-each`

Like `map`, `for-each` applies a procedure to each element of a list, or to corresponding elements of a set of lists. Unlike `map`, `for-each` discards the values returned by all of the applications except the last, and returns the last value. (The applications are also guaranteed to occur in front-to-back list order.) This is sort of like what a `begin` expression does, except that the "subexpressions" are not textually written out—they're applications of a first-class procedure to list items.

Like `begin`, `for-each` is used to execute expressions in sequence, for effect rather than value, except that the last value may be useful.

[ give code ]

### 3.7.6 member and assoc, and friends

The standard Scheme procedures `member` and `assoc` are used for searching lists. I'll show how they can be implemented in Scheme, even though every Scheme system includes them.

Each of these procedures has two alternative versions, which use different equality tests (`eq?` or `eqv?`) when searching for an item in list.

#### 3.7.6.1 member, memq, and memv

`member` searches a list for an item, and returns the remainder of the list starting at the point where that item is found. (That is, it returns the pair whose `car` refers to the item.) It returns `#f` if the item is not in the list.

For example, `(member 3 '(1 4 3 2))` returns `(3 2)`, and `(member 'foo '(bar baz quux))` returns `#f`.

Lists are often used as sets, and `member` serves nicely as a test of set membership. If an item is not found, it returns `#f`, and if it is, it returns a pair. Since a pair is a true value, the result of `member` can be used like a boolean in a conditional.

Since member returns the "rest" of a list, starting with the point where the item is found, it can also be particularly useful with ordered lists, by skipping past all of the elements up to some desired point, and returning the rest.

```
(define (member thing lis)
   (if (pair? lis)
       (if (equal? (car lis) thing)
           lis
           (member thing (cdr lis)))))
```

Note that `member` uses the `equal?` test (data structure equivalence) when searching. This makes sense in situations where you want same-structured data structures to count as "the same." (For example, if you're searching a list of lists, and you want a sublist that has the same structure as the target list to count as "the same.") Note that if the elements of the list are circular data structures, `member` may loop infinitely.

If you want to search for a particular object, you should use `memq?`, which is like `member` except that it uses the `eq?` test, and may be much faster.

If the list may include numbers, and you want copies of the same number to count as "the same", you should use `memv`.

### 3.7.6.2 `assoc`, `assq`, and `assv`

`assoc` is used to search a special kind of nested list called an *association* list. Association lists are often used to represent small tables.

An association list is a list of lists. Each sublist represents an association between a key and a list of values. The car of the list is taken as the key field, but the whole list of values is returned.

(Typically, an association list is used as a simple table to map keys to single values. In that case, you must remember to take the `cadr` of the sublist that `assoc` returns.)

Some example uses:

```
Scheme>(assoc 'julie '((paul august 22) (julie feb 9) (veronique march 28)))
(julie february 9)

Scheme>(assoc 'key2 '((key1 val1) (key2 val2) (key0 val0)))
(key2 val2)

Scheme>(cadr (assoc 'key2 '((key1 val1) (key2 val2) (key0 val0))))
val2

Scheme>(assoc '(feb 9)
              '(((aug 1) maggie phil) ((feb 9) jim heloise) ((jan 6) declan)))
((feb 9) jim heloise)
```

And the code:

```
(define (assoc thing alist)
   (cond ((pair? alist))
          (if (equal? (car (car alist)) thing)
              (car alist)
              (assoc thing (cdr alist)))
         (#t #f)))
```

Like `member`, `assoc` uses the `equal?` test when searching a list. This is what you want if (and only if) you want same-structured data structures to count as "the same."

`assq` is like `assoc`, but uses the `eq?` test. This is the most commonly-used routine for searching association lists, because symbols are commonly used as keys for association lists. (The name `assq` suggests "associate using the `eq?` test.")

If the keys may be numbers, `assv?` should probably be used instead. It considers `=` numbers the same, but otherwise tests object identity, like `eq?`. (The name `assv` suggests "associate using the `eqv?` test.")

### 3.7.7 Procedure Specialization, Composition, and Currying

#### 3.7.7.1 Procedure Specialization

Suppose that we are writing a program where we need to take a list of numbers and produce a corresponding lists with numbers ten times as big.

Notice that we already have a procedure, `map`, that can iterate over a list, apply a function to each item, and return the list of function values. We also have a multiplication procedure, `*` that can multiply numbers by any value we want.

We can't just write `(map * some-list)`, though, because when `map` iterates over a single list, it expects a procedure that takes exactly one argument, and `*` takes *two* arguments. Somehow, we need to supply the argument `10` to each of the calls `map` makes to `*`.

What we need is a one-argument function that multiplies its argument by ten. We could define our own multiplication-by-ten procedure, `*10`, and then use `map` to apply it to the elements of `some-list`.

```
(define (*10 number)
   (* 10 number))

(map *10 some-list)
```

Here we've specialized `*` to create `*10`—we've taken a function with some number of arguments, and produced a function with fewer arguments, which is equivalent to calling the original procedure with the missing argument always the same.

If `*10` is only used in one place, there's really no need to create a named procedure—we can just use a `lambda` expression to create the procedure where we need it, at the call to map:

```
(map (lambda (number)
        (* 10 number))
     some-list)
```

Here we create an anonymous procedure that multiplies its argument by 10, and pass that procedure and a list to `map`, which will map the procedure over the list and return the corresponding list of results.

It is often a good idea to design procedures with specialization in mind.

Consider the similarities between `member`, `memv`, and `memq`. All of them do almost the same thing, with the difference being which equality test they use during a search.

We can define a general procedure, `mem`, which expresses the similarities between these procedures, and then specialize that procedure.

Our general procedure will look like `member`, except that it will take an argument saying which test to use. In Scheme, this is easy—we can simply hand it a first-class procedure like `equal?` or `eq?`, or any other test we want to use, and have it call that procedure to perform the test.

```
(define (mem test-proc thing lis)
   (if (pair? lis)
       (if (test-proc (car lis) thing)
           lis
           (mem test-proc thing (cdr lis)))))
```

To get the effect of `(member some-key some-list)`, we can write `(mem equal? some-key some-list)`.

Note that here we're not calling `equal?`, we're just passing the value of the variable `equal?` (i.e., the procedure first-class procedure object `equal?`) to `mem`. `mem` receives this value when the argument variable `test-proc` is bound, and can call it by that name.

(In the `*10` example, we specialized `*` with data—the number `10`—but here we're specializing `mem` with a procedure. The same technique works, because procedures are data objects, and can be passed as arguments like any other data, then called as procedures.)

If Scheme didn't provide `member`, we could easily define it by specializing `mem`—we simply define `member` to call `mem`, but always pass `equal?` as the first argument:

```
(define (member thing lis)
   (mem equal? thing lis))
```

Likewise, we could define `memq` and `memv` by specializing `mem` with `eq?` and `eqv?`, respectively.

This kind of function specialization is particularly useful when you have a pattern for a procedure, but may need arbitrary variants of it in the future.

For example, suppose you want to search a list of lists, and you want your search routine to return the first sublist whose first two elements match a particular two-element list. (This might be an ordered list of birthdays, and you could be searching for the part of the list that starts with a particular month of a particular year.)

You might search the list for any list whose first elements are `1964` and `December`, by handing it a target list `(1964 December)`, like this:

```
(mem-first-two? '(1964 December)
                '((1961 January 15 "Susan")
                  (1964 March 28 "Edward")
                  (1964 March 29 "Selena")
                  (1964 December 31 "Anton")
                  (1965 January 8 "Booker"))))
```

and get back the result

```
((1964 December 31 "Anton")
 (1965 January 8 "Booker"))))
```

`member`, `memq`, and `memv` are useless for this, but it's pretty easy with `mem`. First we define a match predicate for our purpose:

```
(define (first-two-eqv? target thing)
   (and (eqv? (car target) (car thing))
        (eqv? (cadr target) (cadr thing))))
```

Then we curry `mem` with that predicate to create our search procedure:

```
(define (mem-first-two? thing lis)
   (mem first-two-eqv? thing lis))
```

If `first-two-eqv?` is only likely to be used in `mem-first-two`, we can put it inside `mem-first-two`, as a local procedure, instead of leaving it hanging out where it can be called from other procedures. This is a good idea for a procedure that is so specialized that it's unlikely to be useful in any other way—especially if you're sure it works for what you designed it for, but think it may be tricky to use for slightly different purposes. (For example, we've chosen to use the `eqv?` test for matching list elements, but for some purposes this might be the wrong choice.)

```
(define (mem-first-two thing lis)
   (let ((first-two-eqv? (lambda (target thing)
                           (and (eqv? (car target) (car thing))
                                (eqv? (cadr target) (cadr thing))))))
      (mem first-two-eqv? thing lis)))
```

In this routine, `first-two-eqv?` is only called from one place—the call to `mem`. Rather than defining it as a named procedure, using `letrec` and `lambda`, we can simply use the `lambda` expression at the one place the procedure is needed:

```
(define (mem-first-two thing lis)
   (mem (lambda (target thing)
           (and (eqv? (car target) (car thing))
                (eqv? (cadr target) (cadr thing))))
        target
        lis))
```

This idiom is very common in situations where you need a small procedure in exactly one place.

Likewise, if `mem-first-two` itself is only useful in one place, it would be reasonable to avoid making it a procedure at all, and instead to simply call `mem` from that place:

```
...
(mem (lambda (target thing)
        (and (eqv? (car target) (car thing))
             (eqv? (cadr target) (cadr thing))))
     target
     lis)
...
```

## 3.8 Discussion and Review

# 4 Writing an Interpreter

In this Chapter, I'll show a simple interpreter for a subset of Scheme, written in Scheme.

I'll start out with a very simple interpreter for a tiny subset of Scheme, which only understands simple arithmetic expressions.

Then I'll improve the interpreter in variety of ways.

In a later chapter, we'll return to this interpreter and add macros, [blah blah blah... ]

## 4.1 Interpretation and Compilation

Programming languages are usually implemented by interpreters or compilers, or some mix of both. In reality, almost all language implementations are a mix of both, to at least a small degree, and the line between them is surprisingly fuzzy.

A pure interpreter reads the source text of a program, analyzes it, and executes it as it goes. This is usually very slow—the interpreter spends a lot of time analyzing strings of characters to figure out what they mean. A pure interpreter must recognize and analyze each expression in the source text each time it is encountered, so that it knows what to do next. This is pretty much how most command shell languages work, including UNIX shells and Tcl.

A pure compiler reads the source text of a program, and translates it into machine code that will have the effect of executing the program when it is run. A big advantage of compilers is that they can read through and analyze the source program *once*, and generate code that you can run to give the same effect as interpreting the program. Rather than analyzing each expression each time they encounter it, compilers do the analysis once, but record the actions an interpreter *would* take at that point in the program.

In effect, a compiler is a weird kind of interpreter, that "pretends" to interpret the program, and records what an interpreter would do. It then goes through its record of actions the interpreter would take, and spits out instructions whose effect is the same as what the interpreter would have done. Most of the decision-making that the interpreter does—like figuring out that an expression is an assignment expression, or a procedure call—can be done at compile time, because the expression is the same each time it's encountered in running the program.

The compiler's job is to do the work that's always the same, and spit out instructions that will do the "real work" that can only be done at runtime, because it depends on the actual data that the program is manipulating. For example, an `if` statement is always an `if` statement each time it's encountered, so that analysis can be done once. But which branch will be taken depends on the runtime value of an expression, so the compiler must emit code to test the value of the expression, and take the appropriate branch.

Most real interpreters are somewhere in between pure interpreters and compilers. They read through the source code for a program once, and translate it into an "intermediate representation" that's easier to work with—a data structure of some kind—and then interpret that. Rather than stepping through strings of source text, they step through a data structure that represents that source text in a more convenient form, which is much faster to operate on. That is, they do some analysis once, while converting the source text into a data structure, and the rest as they execute the program by stepping through the data structure.

In this chapter, I'll demonstrate Scheme programming by showing a simple a simple interpreter for a small subset of Scheme, in Scheme. In the next chapter, I'll present a slightly fancier interpreter that implements all of the really important parts of Scheme. There are four good reasons for using a Scheme interpreter as an example Scheme program:

1. A simple interpreter really is simple, but it can show off some of the handy features of Scheme. It's a good example of Scheme programming.
2. Most serious programs include some kind of command interpreter, so every programmer should know how to write a decent one. Often, the command interpreter has a tremendous impact on the usability and power of a system, and too many programs have bad ones.
3. Understanding how a Scheme interpreter works may clarify language issues. It gives you a nice, concrete, understanding of what Scheme does when it encounters an expression, so you know what your programs will do—it'll be obvious when you need a quote, or parentheses, and when you don't.
4. Every programmer should understand the basics of how a compiler works. Understanding a Scheme interpreter gets you half-way to understanding a Scheme compiler. A Scheme compiler is really very much like a Scheme interpreter—it analyzes Scheme expressions and figures out what to do. The main difference between an interpreter and a compiler is just that when an interpreter figures out what to do, it does it immediately, while a compiler records what to do when you run the program later.

## 4.2 Implementing a Simple Interpreter

In this section, we'll use Scheme to implement an interpreter for a tiny subset of Scheme—just simple arithemetic expressions. The interpreter we'll show is simple, but it's a real interpreter—it works on the same principles as many real Scheme systems. In the next chapter, we'll show how a slightly more complicated interpreter which implements most of Scheme's important features, and the skeleton of a compiler for Scheme.

The interpreter is a good example for learning Scheme programming, because it makes heavy use of recursion—the processes of reading and evaluation are naturally recursive. As you'll see, the code is also an example of mostly-functional programming (with very few side effects); using recursion in the natural way avoids the need for side effects, because data structures are generally created at the right times, rather than being created too early and having to be updated later.

Our interpreter will use Scheme's built-in `read` procedure to accept input in the form of *s-expressions*, i.e., expressions represented as standard Scheme data structures such as symbols, numbers, and possibly nested lists of those consituents. [Recall that...] S-expressions can be simple, as in the case of symbols, or complex, as in the case of nested lists.

### 4.2.1 The Read-Eval-Print Loop

(This section could be skimmed if you're not interested in the read-eval-print-loop, which is just a simple command interpreter that acts as a "front end" to the evaluator.)

When you're interacting with Scheme by typing text, you're interacting with a Scheme procedure called the *read-eval-print* loop. This procedure just loops, accepting one command at a time, executing it, and printing the result.

The three steps at each iteration of the loop are:

1. calling `read` to an expression from the keyboard input buffer, constructing a data structure to represent it,
2. calling `eval` interpreter to "interpret" the expression,
3. calling `write` to print the resulting value so that the user can see it.

You can write your own read-eval-print loop for your own programs, so that users can type in expressions, and you can interpret them any way you want. Later, I'll show how to write an interpreter, and this will come in handy. You can start up your read-eval-print loop (by typing in `(rep-loop)`), and it will take over from the normal Scheme read-eval-print loop, interpreting expressions your way.

Here's a very simple read-eval-print loop:

```
(define (rep-loop)
   (display "repl>")      ; print a prompt
   (write (eval (read)))  ; read expr., pass to eval, write result
   (rep-loop))            ; tail call to do it again
```

We've coded the iteration recursively, rather than using a looping construct. The procedure is tail-recursive, since all it does at the end is call itself. Remember that Scheme is smart about this kind of recursion, and won't build up procedure activation information on the stack and cause a stack overflow. You can do tail recursion all day. Since nothing happens in a given call to the procedure after the tail-call, Scheme can avoid returning to it at all, and avoid saving any state to return to.

The above read-eval-print loop isn't very friendly, because it loops infinitely without giving you any chance to break out of it. Let's modify it to allow you to stop the tail recursion by typing in the symbol `halt`.

```
(define (rep-loop)
   (display "repl>")              ; print a prompt
   (let ((expr (read)))           ; read an expression, save it in expr
      (cond ((eq? expr halt)      ; user asked to stop?
             (display "exiting read-eval-print loop")
             (newline))
            (#t                   ; otherwise,
             (write (eval expr))  ;  evaluate and print
             (newline)
             (rep-loop)))))       ;  and loop to do it again
```

Notice that this is still tail recursive, because the branch that does a the recursive call doesn't do anything else after that.

This read-eval-print loop could be improved a little. By using the symbol `halt` as the command to tell the loop to stop, we prevent people from being able to evaluate `halt` as an expression. We could get around this by ensuring that the halt command doesn't have the syntax of any expression in the language, but we won't bother right now.

Another improvement would be to make it possible to use different interpreters with the same read-eval-print loop. The `rep-loop` procedure above assumes that it should call a procedure named `eval` to evaluate an expression. We'd like to write a `rep-loop` that works with different evaluators, so instead of having it call `eval` by name, we'll hand it an argument saying which evaluator to use. Since procedures are first class, we can just hand it a pointer to the evaluation procedure.

```
(define (rep-loop our-eval)
   (display "repl>")                ; print a prompt
   (let ((expr (read)))             ; read an expression, save it in expr
      (cond ((eq? expr 'exit)       ; user asked to stop?
             (display "exiting read-eval-print loop")
             (newline))
            (#t                     ; otherwise,
             (write (our-eval expr)) ;  evaluate and print
             (rep-loop our-eval))))) ;  and loop to do it again
```

Here we just made three changes. We added an argument `our-eval`, which is expected to be a procedure. Then we changed the call to `eval` to call `our-eval`, i.e., whatever evaluator was given. Then we changed the recursive call to `rep-loop` to pass that argument on to the next recursive call.

### 4.2.2 The Reader

(Note: this section could be skimmed if you're not interested in how the reader works—it is just a front end to the evaluator, where the interesting work is done.)

We won't write our own reader for our interpreter, but I'll sketch how the reader works.

(Our interpreter will just snarf the reader from the underlying Scheme system we're implementing it in, but it's good to know how we could write a reader, and it's a nice example of recursive programming.)

The reader is just the procedure `read`, which is written in terms of a few lower-level procedures that read individual characters and construct *tokens*, which `read` puts together into nested data structures. A token is just a fairly simple item that doesn't have a nested structure. For example, lists nest, but symbol names don't, strings don't, and numbers don't.

The low-level routines that `read` uses just read individual tokens from the input (a stream of characters). These tokens include symbols, strings, numbers, and parentheses. Parentheses are special, because they tell the reader when recursion is needed to read nested data structures.

(I haven't explained about character I/O, but don't worry—there are Scheme procedures for reading a character of input at a time, testing characters for equality, etc. For now, we'll ignore those details and I'll just sketch the overall structure of the reader.)

Lets assume we have a simple reader that only reads symbols, integers, and strings, and (possibly nested) lists made up of those things. It'll be pretty clear how to extend it to read other kinds of things.

#### 4.2.2.1 Implementing `read`

`read` uses recursion to construct nested lists while reading through the character input from left to right.

When it sees a left parenthesis, it calls an auxiliary procedure we'll call `read-list` to read the elements of a list.

`read` and `read-list` are mutually recursive. `read-list` reads the elements of a list by calling `read` (if the list elements are simple tokens), which may call `read-list` recursively to read nested lists.

Notice that hitting a right parenthesis is the termination condition for the recursion. If we're reading a sublist of a list, and hit a right parenthesis, `read-list` recognizes that as the sign to stop, and return a complete (nested) list to `read`.

Here's a slightly oversimplified version of `read`. (The main oversimplification is that we've left out any error-checking code. We assume that what we're reading is a legal textual representation of a Scheme data structure. We also haven't dealt with reading from files, instead of the standard input, or what to do when reaching the end of a file.)

(Our little reader will use the standard Scheme procedure `read-char` to read one character of input at a time, and also the predictate procedures `char-alphabetic?` and `char-digit?`; these tell whether a character represents a letter or a number. We'll also use the character literals #\" and #\(, which represent the double quote character and the left parenthesis character.)

[ this code is off the top of my head and needs to be debugged ]

```
(define (read)
   (let ((first-char (read-char)))
      (if (eq? first-char #\()   ; char a left parenthesis?
          (read-list)
          (cond ((char-alphabetic? first-char)
                 (read-symbol first-char))
                ((char-digit? first-char)
                 (read-number first-char))
                ((eq? first-char #\")  ; char a double quote?
                 (read-string))))))
```

Notice that the first `if` is the recursion test. If we see a left parenthesis, which is a special token, and we call `read-list` to read a list. `read list` can call `read` again to read the elements of the list, so `read` is indirectly recursive.

If we're not reading a list, we call any of several auxiliary procedures to read tokens:

`read-symbol`. If the character we read is a letter, we're reading a symbol, so we call `read-symbol` to finish reading it. (We pass it the character we read, since it's the first character of the symbol's print name.) `read-symbol` (not shown) just reads through more characters, saving them until it hits a special token (space or parenthesis). When it finishes reading the whole print name of the symbol, it checks the table of symbols to see if there's already a symbol by that name. If so, it just returns a pointer to it. If not, it constructs a symbol by that name, adds it to the table, and returns a pointer to that.

`read-number`. If the character we read is a digit, we're reading a number, so we call `read-number`. (We pass it the first character we read, since that's the first digit of the number.) `read-number` just reads through successive characters, saving them until it hits a special token such as a space or parenthesis. Then it calls another procedure, `string->number` which converts the sequence of digit characters into a binary number in the usual Scheme number representation, and returns that.

`read-string`. If the character we read is a double quote ("), we're reading a string, so we call `read-string`. (We don't have to pass it the character we read, since the double quote isn't actually part of the string. It's done its job by telling us that we're reading a string.)

`read-string` just reads through characters, saving them until it hits another double quote. It then calls another procedure that constructs a string with that sequence of characters.

#### 4.2.2.2 Implementing `read-list`

Here's a slightly oversimplified version of `read-list`. (Again, the main oversimplification is that we don't check for illegal syntax, like extra closing parentheses.)

[ need to explain peek-character ]

[ this code is off the top of my head and needs to be debugged ]

```
(define (read-list list-so-far)
   (let ((next-char (peek-char)))
      ; if we hit a right parenthesis,
      (if (eq? next-char #\))
          ; then return list we've read, reversing it into proper order
          (reverse list-so-far)
          ; else read next item and call self recursively to read rest
          (cons (read)
                (read-list list-so-far)))))
```

Notice that we've coded `read-list` recursively in two ways.

We code the iteration that reads successive items in the list as a recursion, passing the list so far as an argument to the recursive call. [ This is not tail-recursive, but we could fix it. ]

We read list elements by calling `read`, and then cons them onto the list so far, and pass that to a recursive call to `read-list`. This constructs a list that's backwards, because we push later elements onto the front of the list. When we hit a right parenthesis and end a recursive call, we reverse the list we've read, to put it in the proper order.

#### 4.2.2.3 Comments on the Reader

The reader is really a simple kind of *recursive descent parser*. A *parser* converts a sequence of tokens into a syntax tree that describes the nesting of expressions or statements.) It is a "top-down" parser, because it recognizes high-level structures before lower-level ones (e.g., it recognizes the beginning of a list before reading and recognizing the tokens and sublists inside it).[12]

It converts a linear sequence of characters into a simple *parse tree*.

(If you're familiar with standard compiler terminology, you should recognize that `read` performs lexical analysis (a.k.a. scanning or tokenization) using `read-string`, `read-symbol`, and `read-number`. It performs `predictive` recursive-descent parsing via the mutual recursion of `read` and `read-list`.)

Unlike most parsers, the data structure `read` generates is a data structure in the Scheme language, rather than a data structure internal to a compiler or interpreter. This is one of the nice things about Scheme—there's a simple but flexible parser you can use in your own programs. You can use it for parsing data as well as programs.

When implementing the Scheme language, that's not all there is to doing parsing. The reader does the first part of parsing, translating input into s-expressions. The rest of parsing is done during interpretation or compilation, in a very straightforward way. The rest of the parsing isn't much more complicated than reading, and is also done recursively.[3]

---

[1] Unsurprisingly, a *bottom-up* parser would do the opposite—it would recognizes the smaller consituents first, and then recognizes the larger groupings that enclose them.

[2] In the technical terminology of programming language processors, the reader is a *predictive* parser for an *LL* grammar. It can parse s-expressions top-down in a single pass through the sequence of tokens, without looking ahead more than one token, because it only needs to see the next token to know what action to take. (E.g., if it sees a left parenthesis, it immediately "knows" that it is parsing a nested list.)

[3] It's often said that Lisp and Scheme have such a simple syntax that they "don't need a parser," but this is just false. Lisp and Scheme actually have \em two parsers, because their syntax has a two levels. The "surface" syntax is parenthesized prefix expressions, recognized by the reader, but there is a "deeper" syntax that is recognized by the interpreter or compiler, which analyzes s-expressions in the process of evaluating or compiling them.

As we'll see when we get to macros, Scheme syntax is even more sophisticated than this, despite its simplicity. Technically, Scheme has a \em transformational grammar that is not "context-free," but is easy to parse. (If you don't know what that means, don't worry about it. Scheme is easy to understand without knowing the fancy technical terms.)

### 4.2.3 Recursive Evaluation

The evaluator is the core of the interpreter—it's what does all of the interesting work to evaluate complicated expressions.

Evaluation is done recursively. We write code to evaluate simple expressions, and use recursion to break down complicated expressions into simple parts.

I'll show you a simple evaluator for simple arithmetic expressions, like a four-function calculator, which you can use like this, given the read-eval-print-loop above:

```
Scheme>(repl math-eval)  ; start up read-eval-print loop w/arithmetic eval
repl>1
1
repl>(+ 1 2)
3
repl>(* (+ 1 3) (- 4 2))
8
```

As before, the read-eval-print-loop reads what you type at the `repl>` prompt as an s-expression, and calls `math-eval`.

Here's the main dispatch routine of the interpreter, which figures out what kind of expression it's given, and either evaluates it trivially or calls `eval-combo` to help:

```
(define (math-eval expr)
  (cond ;; self-evaluating object?  (we only handle numbers)
        ((number? expr)
         expr)
        ;; compound expression? (we only handle two-arg combinations)
        ((and (list? expr)
              (eq? (length expr) 3))
         (math-eval-combo expr))
        (#t
         (error "invalid expression" expr))))
```

First `math-eval` checks the expression to see if it's something simple that it can evaluate straightforwardly, without recursion.

The only simple expressions in our language are numeric literals, so `math-eval` just uses number? to test whether the expression is a number. If so, it just returns that value. (Voila! We've implemented self-evaluating objects.)

If the expression is not simple, it's supposed to be an arithmetic expression with an operator and two operands, represented as a three element list. (This is the subset of Scheme's combinations that this interpreter can handle.) In this case, `math-eval` calls `eval-combo`.

```
(define (math-eval-combo expr)
  (let ((arg1 (math-eval (cadr expr)))
        (arg2 (math-eval (caddr expr))))
     (cond ((eq? (car expr) '+)
            (+ arg1 arg2))
           ((eq? (car expr) '-)
            (- arg1 arg2))
           ((eq? (car expr) '*)
            (* arg1 arg2))
           ((eq? (car expr) '/)
            (/ arg1 arg2))
           (#t
            (error "Invalid operation in expr:" expr)))))
```

`eval-combo` handles a combination (math operation) by calling `math-eval` recursively to evaluate the arguments, checking which operator is used in the expression, and calling the appropriate Scheme procedure to perform the actual operation.

### 4.2.4 A Note on Snarfing and Bootstrapping

Two concepts worth knowing about language implementation are `snarfing` and `bootstrapping`. Snarfing is "stealing" features from an underlying language when implementing a new language. Bootstrapping is the process of building a language implementation (or other system) by using the system to extend itself.

#### 4.2.4.1 Snarfing

Our example interpreter implements Scheme in Scheme, but we could have written it in C or assembly language. If we had done that, we'd have to have written our own read-eval-print loop, and a bunch of not-very interesting code to read from the keyboard input and create data structures, display data structures on the screen, and so on. Instead, we "cheated" by snarfing those features from the underlying Scheme system—we simply took features from the underlying Scheme system and used them in the language we interpret. Our tiny language requires you to type in Scheme lists, because it uses the Scheme read-eval-print to get its input and call the interpreter. If we wanted to, we could provide our own read routine that reads things in a different syntax. For example, we might read input that uses square brackets instead of parentheses for nesting, or which uses infix operators instead of prefix operators. (That is, the middle item in a three-element list would be the operator name.)

There are some features we *didn't* just snarf, though—we wrote our own evaluation procedure which controls recursive evaluation. For example, we *use* basic Scheme arithemetic procedures to implement individual arithmetic operations, but we don't simply *snarf* them: the interpreter recognizes arithmetic operations in its input language, and maps them onto procedure calls in the underlying language. We can change our language by changing those mappings: for example, we could use the symbols `sum`, `difference`, `product`, and `quotient` instead of `+`, `-`, `*`, and `*`. Or we could use the same names, but implement the operations differently. (For example, we might have our own arithmetic routines that allow a representation of infinity, and do something reasonable for division by zero.)

We also use recursion to implement recursion, when we recursively call `eval`). But since we coded that recursion explicitly, we can easily change it, and do something different. Our arithmetic expressions don't have to have the same recursive structure as Scheme expressions.

We could also implement recursion ourselves. As written, our tiny interpreter uses Scheme's recursion "stack" to implement it's own stack—each recursive call to `eval` implements a recursive call in our input language. We didn't have to do this. We could have implemented our own stack as a data structure, and written our interpreter as a simple non-recursive loop.

What counts as "snarfing"? The term is a good one, but not clearly defined. We clearly just snarfed the Scheme reader, but we've done something a little different with recursion. We've done something very different with the interpretation of operator names.

#### 4.2.4.2 Bootstrapping and Cross-compiling

Implementing a programming language well requires attention to the fine art of bootstrapping—how much of the system do you have to build "by hand" in some lower-level system, and how much can you build *within the system itself*, once you've got a little bit of it working.

Most Scheme systems are written mostly in Scheme, and in fact it's possible (but not particularly fun) to implement a whole Scheme system in Scheme, even on a machine that doesn't have a Scheme system yet.

How are these things possible?

First, let's take the simple case, where you're willing to write a little code in another language. You can write an interpreter for a small subset of Scheme in, say, C or assembler. Then you can extend that little language by writing the rest of Scheme in Scheme—you just need a simple little subset to get started, and then things you need can be defined in terms of things you already have. Writing an interpreter for a subset of Scheme in C is not hard—just a little tedious. Then you can use `lambda` to create most of the rest of the procedures in terms of simpler procedures. Interestingly, you can also implement most of the defining constructs and control constructs of Scheme in Scheme, by writing *macros*, which we'll discuss later.

You can start out this way even if you want your Scheme system to use a compiler. You can write the compiler in Scheme, and use the interpreter to run it and generate machine code. Now you have a compiler for Scheme code, and can compile procedures so that they run faster than if you interpreted them. You can take most of the Scheme code that you'd been interpreting, and use the compiler to create faster versions of them. You then replace the old (interpreted) versions with the new (compiled) versions, and the system is suddenly faster.

Once the compiler works, you can *compile the compiler*, so that \em it runs faster. After all, a compiler is just a program that takes source code as input and generates executable code—it's just a program that happens to operate on programs. Now you're set—you have a compiler that can compile Scheme code that you need to run, including itself, and you don't need the interpreter anymore.

To get Scheme to work on a new system, without even needing an interpreter, you can \em cross-compile. If you have Scheme working on one kind of machine, but want to run it on another, you can write your Scheme compiler in Scheme, and have it run on one machine but generate code for the new machine. Then you can take the executable code it generates, copy it onto the new machine, and run it.

Most Scheme systems are built using tricks like this. For example, the RScheme system never had an interpreter at all. Its compiler was initially run in a different Scheme system (Scheme-48) and used to compile most of RScheme itself. This code was then used to run RScheme with no further assistance from another implementation.

The first Scheme system was built by writing a Scheme interpreter in Lisp, [ or was it a compiler first? ... blah blah ... ]

### 4.2.5 Improving the Simple Interpreter

We can easily improve the little interpreter in lots of ways. [ We should put the code in a file minieval.scm so people can experiment with it. Need to debug it first, of course. It's changed since the one I've used in class. ]

First, we can add a toplevel binding environment, so we can have some variables. (Local variables will be discussed in the next chapter.) To make them useful, we need some special forms, like `define` and (while we're at it) `set!`.

We can also add a few more data types; for now, we'll just add booleans.

Here's what our new main dispatch routine looks like:

```
(define (eval expr)
   (cond ;; variable reference
         ((symbol? expr)
          (eval-symbol expr))
         ;; combination OR special form
         ((pair? expr)
          (eval-list expr))
         ;; any kind of self-evaluating object
         ((self-evaluating? expr)
          expr)
         (#t
          (error "Illegal expression: " expr))))
```

Since we're adding variables to our interpreter, symbols can be expressions by themselves now—references to top-level variable bindings. We've added a branch to our `cond` to handle that, and a helper procedure `eval-symbol`. (We'll discuss how the variable lookup is done shortly.)

We need to recognize two kinds of self-evaluating types now (and may add more later), so we come up with a procedure `self-evaluating?` that covers both cases and can easily be extended.

```
(define (self-evaluating? expr)
   (or (number? expr) (boolean? expr)))
```

We also need to recognize two basic types of compound expressions: combinations and special forms. These (and only these) are represented as lists, so we can use `pair?` as a test, and dispatch to `eval-list`.

Here's the code for `eval-list`, which just checks to see whether a compound expression is a special form, and dispatches to `eval-special-form` if it is, and `eval-combo` if it's not.

```
(define (eval-list expr)
   (if (and (symbol? (car expr))
            (special-form-name? (car expr)))
       (eval-special-form expr)
       (eval-combo)))
```

We could use a `cond` to check whether symbols are special form names, but using `member` on a literal list is clearer and easily extensible—you can just add names to the list.

```
(define (special-form-name? expr)
   (member '(if define set!)))
```

`eval-special-form` just dispatches again, calling a routine that handles whatever kind of special form it's faced with. (Later, we'll see prettier ways of doing this kind of dispatching, using first-class procedures.) From here, we've done most of the analysis, and are dispatching to little procedures that actually do the work.

[ need to come back to this after discussing backquote—this would make a good example ]

```
(define (eval-special-form expr)
  (let ((name (car expr)))
     (cond ((eq? name 'define)
            (eval-define expr))
           ((eq? name 'set!)
            (eval-set! expr))
           ((eq? name 'if)
            (eval-if expr)))))
```

Once the evaluator has recognized an `if` expression, it calls `eval-if` to do the work. `eval-if` calls `eval` recursively, to evaluate the condition expression, and depending on the result, calls it again to evaluate the "then" branch or the "else" branch. (One slight complication is that we may have a one-branch else, so `eval-if` has to check to see if the else branch is there. If not, it just returns `#f`.)

```
(define (eval-if expr)
   (let ((expr-length (length expr)))
      (if (eval (cadr expr))
          (eval (caddr expr))
          (if (= expr-length 4))
              (eval (cadddr expr))
              #f)))
```

[ note that what we're doing includes parsing... one-branch vs. two branch `if`. Should actually be doing *more* parsing, checking syntax and signaling errors gracefully. E.g., should check to see that `expr-length` is a legal length. ]

[ Also note that we're snarfing booleans, and our `if` behaves like a Scheme `if`... but we don't have to. We could put a different interpretation on `if`, e.g., only interpreting `#t` as a true value. ]

For a toplevel binding environment, we'll use an association list. (A more serious interpreter would probably use a hash table, but a association list will suffice to demonstrate the principles.)

We start by declaring a variable to hold our interpreter's environment, and initializing it with an empty list.

```
(define envt '())
```

To add bindings, we can define a routine to add an association to the association list.

```
(define (toplevel-bind! name value)
   (let ((bdg (assoc name toplevel-envt)))
      ;; if binding already exists, put new value (in cadr) of association
      ;; else create a new association with given value
      (if bdg
          (set-car! (cdr bdg) value)
          (set! envt
                (cons (list name init) envt)))))
```

Recall that the elements of an association list are "associations," which are just lists whose first value is used as a key. We'll use the second element of the list as the actual storage for a variable.

For example, an environment containing just bindings of `foo` and `bar` with values `2` and `3` (respectively) would look like `((foo 2) (bar 3))`.

At the level of the little Scheme subset we're implementing, we'd draw this environment this way:

```
     +--------+       [envt]
envt |   *---+------>+--------+
     +--------+  foo |   *---+---> 2
                     +--------+
                 bar |   *---+---> 3
                     +--------+
```

This emphasizes the fact that these are variable bindings with values, i.e., named storage locations. Notice that `envt` is a variable in the language we're using to implement our interpreter, but `foo` and `bar` are variables in the language the interpreter implements.

If we want to show how it's implemented at the level of the Scheme we're writing our interpreter in, we can draw it more like this:

```
     +--------+
envt |   *---+---->+---+---+                   +---+---+
     +--------+    | * | *-+------------------>| * | * +
                   +-|-+---+                   +-+-+---+
                     |                           |
                   +---+---+   +---+---+       +---+---+   +---+---+
                   | * | +-+-->| * | * |       | * | *-+-->| * | * |
                   +-|-+---+   +-|-+---+       +-|-+---+   +-+-+---+
                     |           |               |           |
                    foo          2              bar          3
```

Now we can add the four procedures we had in the math evaluator:

```
(toplevel-bind! '+ +)
(toplevel-bind! '- -)
(toplevel-bind! '* *)
(toplevel-bind! '/ /)
```

Again, we're just snarfing procedures straight from the Scheme we're implementing our interpreter in. We put them in our binding environment under the same names.

Now we need accessor routines to get and set values of bindings for variable lookups and `set!`

```
(define (toplevel-get name)
   (cadr (assoc name envt)))

(define (toplevel-set! name)
   (set-car (cdr (assoc name envt))))
```

[ of course, these really should have some error checking ]

Given this machinery, we can now write `eval-define` and `eval-set!`. All they do is extract a variable name from the `define` or `set!` expression, and create binding for that name or update its value.

```
(define (eval-define expr)
   (toplevel-bind! (cadr expr)
                   (eval (caddr expr))))

(define (eval-set! expr)
   (toplevel-set! (cadr expr)
                  (eval (caddr expr))))
```

## 4.3 Discussion and Review

# 5 Environments and Procedures

## 5.1 Understanding `let` and `lambda`

### 5.1.1 `let`

One difference between C or Pascal blocks and Scheme `let`'s is that `let` variable bindings don't necessarily cease to exist when the `let` is exited, and the bindings therefore can't be allocated on a stack in the general case. (The reason for this will become clear when we talk about `lambda` and closures.)

One way to visualize the creation of block variables is to see it as the creation of a new table mapping names to storage, like the toplevel environment in our interpreter.

Except for the new variables, the new environment (table) is the same as the one that was in use when the block was entered. We say that the `let` expression "extends" the "outer" environment with bindings for the `let` variables.

Suppose we type a `let` expression at the Scheme prompt, (Assume we we're just doing the usual expression evaluation in the usual top-level environment.)

```
Scheme>(let ((x 10) (y 20))
         (+ x y))
30
```

The interpreter maintains a pointer to the "current environment" when evaluating an expression. This pointer always points to the environment the currently-executing code must execute in, i.e., the variable bindings it must see for the variables it uses.

Before evaluating the `let` expression, Scheme's environment pointer points to the top-level environment, which contains the usual bindings holding the built-in Scheme procedures, plus any top-level variables we've defined. Supposing we've defined a variable `foo`, we can draw the top-level environement like this:

```
     +-----+        +------+-----+
envt |  +--+------->|  car |  *--+----> #<proc ...>#
     +-----+        +------+-----+
                    | cons |  *--+----> #<proc ...>#
                    +------+-----+
                    |   +  |  *--+----> #<proc ...>#
                    +------+-----+
                    |      *     |
                           *
                    |      *     |
                    +------+-----+
                    |  foo |  +--+----> #<proc ...>#
                    +------+-----+
```

(Here' we've drawn the environment as a simple table of names and bindings. It might actually be implemented as an association list, as in our simple example interpreter, or more likely as a hash table.)

After entering the `let` and creating the bindings for `x` and `y`, the interpreter changes the environment pointer to point to the resulting new environment. This is typically implemented by representing the environment as a chain of tables, rather than a simple table. The newest table is searched first, and so on down the chain, to find the appropriate bindings. This environment chain is used as a pointer-linked stack, for the most part, with new environments being pushed onto the stack when a `let` is entered, and popped off the stack when a `let` is exited.

```
                    +-------+-----+
                    |  car  |  +--+----> #<proc ...>#
                    +-------+-----+
                    | cons  |  +--+----> #<proc ...>#
                    +-------+-----+
                    |   +   |  +--+----> #<proc ...>#
                    +-------+-----+
                    |       *     |
                            *
                    |       *     |
                    +-------+-----+
                    |  foo  |  +--+----> #<proc ...>#
                    +-------+-----+
                              /|\
                               |
                               |
                               |
        +-----+     +-------+--+--+
   envt |  +--+---->|[scope]|  *  |
        +-----+     +-------+-----+
                    |   x   | 10  |
                    +-------+-----+
                    |   y   | 20  |
                    +-------+-----+
```

The link that connects the two tables is called a *scope* link. It reflects the nesting of naming scopes in the program. In this case, when a variable is referenced inside the `let`, the search for a binding begins at the new (small) table. If it is not found, the search follows the scope link to the next table and looks there. This can continue for as many levels as there are nested scopes in the program.

While we're executing in the new environment, its bindings *shadow* (hide) any bindings of variables with the same name in the outer environment. For example, if there's a top-level variable named `x` bound in the top-level environment, they won't be seen by code executing in the `let` environment.

When we exit the `let`, the current environment pointer is set back to point to the same environemnt as before entering the `let`. In the usual case, that environment becomes garbage because there are no pointers to it, and the garbage collector will eventually reclaim its space.

### 5.1.2 lambda

In Scheme, you can create anonymous (unnamed) procedures any time you want, using the `lambda` special form.

[ Say this here, or in Intro chapter?: A better name for `lambda` might be `make-procedure`. ... ]

For example, suppose you want to write a piece of code that needs to double the values of the items in a list. You could do what we did before, and define a named `double` procedure, but if you only need to use the procedue in one place, it's easier to use an anonymous procedure created with `lambda`.

Instead of writing

```
(define (double x)
   (+ x x))
```

and then using it like this

```
   ...
   (map double mylist)
   ...
```

You can simply define it where it's used, using `lambda`.

```
   ...
   (map (lambda (x) (+ x x)) mylist)
   ...
```

This can help avoid cluttering your code with lots of auxiliary procedures. (Don't overdo it, though—if a procedure is nontrivial, it's good to give it a name that reflects what it does.) This is very convenient when using higher-order procedures like `map`, or higher-order procedures you come up with for your own programs.

[ As we'll see in a little while, `lambda` has some very interesting properties that make it more useful than it might seem right now. ]

[ point out that variable arity works with lambda arg lists just like with define arg lists ]

#### 5.1.2.1 `define` and `lambda`

[ Have we already said that procedure define is equivalent to variable define plus lambda? If not, say so here. Either way, maybe explain more clearly here. ]

#### 5.1.2.2 Currying

#### 5.1.2.3 Procedures are Closures

Scheme procedure's aren't really just pieces of code you can execute; they're *closures*.

A closure is a procedure that records what environment it was created in. When you call it, that environment is restored before the actual code is executed. This ensures that when a procedure executes, it sees the exact same variable bindings that were visible when it was created—it doesn't just remember variable names in its code, it remembers what *storage* each name referred to when it was created.

Since variable bindings are allocated on the heap, not on a stack, this allows procedures to remember binding environments even after the expressions that created those environments have been evaluated. For example, a closure created by a `lambda` inside a `let` will remember the `let`'s variable bindings even after we've exited the `let`. As long as we have a pointer to the procedure (closure), the bindings it refers to are guaranteed to exist. (The garbage collector will not reclaim the procedure's storage, or the storage for the `let` bindings.)

Here's an example that may clarify this, and show one way of taking advantage of it.

Suppose we type the following expression at the Scheme prompt, to be interpreted in a top-level environment:

```
Scheme> (let ((count 0))
          (lambda ()
             (begin (set! count (+ count 1))
                    count)))
#<proc ....>#
```

Evaluating this `let` expression first creates a binding environment with a binding for count. The initial value of this binding is 0. In this environment, the lambda expression creates a closure. When executed, this procedure will increment the count, and then return its value. (Note that the procedure is not executed yet, however—it's just created.) This procedure, returned by the lambda expression, is also returned as the value of the let expression, because a `let` returns the value of its last body expression. The read-eval-print loop therefore prints a representation of the (anonymous) procedure.

Unfortunately, we didn't do anything with the value, like give it a name, so we can't refer to it anymore, and the garbage collector will just reclaim it. (OOPS.) Now suppose we want to do the same thing, but hold onto the closure so that we can do something with it. We'll bind a new variable `my-counter`, and use the above `let` expression to create a new environment and procedure, just like before.

```
Scheme> (define my-counter (let ((count 0))
                             (lambda ()
                                (set! count (+ count 1))
                                count))))
my-counter
```

Now we have a top-level binding of `my-counter`, whose value is the procedure we created. The procedure keeps a pointer to the environment created by the `let`, which in turn has a pointer to the top-level environment, thus:

[ should simplify this picture and use it earlier, for the simpler example where we don't keep a pointer to the closure. Should show the envt register pointing to the `let` envt at the moment the closure is created. ]

```
                    [envt]
         +-->+------------+-----+
         |   |      car   |  *--+--> ...
         |   +------------+-----+
         |   |      cons  |  *--+--> ...
         |   +------------+-----+
         |   |        *         |
         |            *
         |   |        *         |
         |   +------------+-----+
         |   | my-counter |  *--+------------+
         |   +------------+-----+            |
         |                  /|\              |
         |                   |               |
         |     [envt]        |               |
         |   +------------+--+--+            |
         |   |   [scope]  |  *  |            |
         |   +------------+-----+            |
         |   |      count |  *--+-->10       |
         |   +------------+-----+           \|/
         |            /|\                [closure]
         |             |                +---------+
         |             +----------------+----*    |
         |                              +---------+
         |                              |    *    |
         |                              +----+----+
         |                                   |
         |                                  \|/
         |                                 [code]
         |                          +--------------------+
     +---+---+                      | (set! count        |
envt |   *   |                      |       (+ count 1)) |
     +-------+                      | count              |
                                    +--------------------+
```

Now if we call the procedure `my-counter`, it will execute in its own "captured" environment (created by the `let`). It will increment the binding of count in that environment, and return the result. The environment will continue to exist as long as the procedure does, and will store the latest value until next time `my-counter` is called:

```
Scheme>(my-counter)
1
Scheme>(my-counter)
2
Scheme>(my-counter)
3
```

Notice that if we evaluate the let form again, we will get a new `let` environment, and a new procedure that will increment and return its `count` value—in effect, each procedure has its own little piece of state which only it can see (because only it was created in that particular environment).

If we want, we can define a procedure that will create new environments, and new procedures that capture those environments—we can generate new counter procedures just by calling that "higher-order" procedure. (Recall that a higher-order procedure is just a procedure that manipulates other procedures. In this case, we're making a procedure that *generates* procedures.)

Each time `make-counter` is called, it will execute a `let`, creating an environment, and inside that it will use `lambda` to create a counter procedure.

```
Scheme> (define (make-counter)
           (let ((count 0))
              (lambda ()
                 (set! count (+ count 1))
                 count)))
make-counter
```

Each of the resulting procedures will have its own captured count variable, and keep it independently of the other procedures.

Make sure you understand that the above procedure definition could have used an explicit `lambda` to create the procedure `make-counter`, rather than the special procedure definition syntax:

```
Scheme> (define make-counter
                (lambda ()
                   (let ((count 0))
                      (lambda ()
                         (set! count (+ count 1))
                         count)))
```

You may actually find this easier to understand, because it shows you exactly what's going on: binding `make-counter` and creating a procedure (with the outer `lambda`) that *when called*, will evaluate a `let` to create an environment, and a `lambda` (the inner one) to create a procedure that captures it.)

Now we'll call the procedure created by the above definition, three times, and each time it will create a new procedure:

```
Scheme> (define c1 (make-counter))
C1
Scheme> (define c2 (make-counter))
C2
Scheme> (define c3 (make-counter))
C3
```

Now we'll call those procedures and look at their return values, to illustrate that they're independent counters:

```
Scheme> (c1)
1
Scheme> (c1)
2
Scheme> (c2)
1
Scheme> (c2)
2
Scheme> (c1)
3
Scheme> (c1)
4
Scheme> (c3)
1
```

Neat, huh? The combination of block structure (local environments) with first-class procedures (closures), allows us to associate state with procedures. Garbage collection makes this very convenient, because we know that the environments will hang around as long as the procedures do.

If you're familiar with object-oriented programming, you may notice a resemblance between closures and "objects" in the object-oriented sense. A closure associates data with a procedure, where an object associates data with multiple procedures. After we get to object-oriented programming, we'll explain how object-oriented programming facilities can be implemented in Scheme using closures.

If you're familiar with graphical user interface systems, you may notice that GUI's often use "callbacks," which are procedures that are executed in response to user input events like button clicks and menu selections, and do something application-specific. (The application "registers" callback procedures with the GUI system, which then calls them when the user clicks on the specified buttons.) Closures make excellent GUI callback procedures, because the application can create a closure for a specific context by capturing variable bindings, to customize the behavior of the procedure.

## 5.2 Lambda is cheap, and Closures are Fast

It may seem that `lambda` is an expensive operation—after all, it creates procedure objects on the fly. At first glance, you might think that executing lambda would require a call to the compiler each time. This is not the case, though, and lambda is actually a fairly cheap constant-time operation.

Notice that the procedure part of a `lambda` expression is known at compile time—each time the `lambda` is executed at run time, it will create a new closure, and may capture a new environment, but the expression closed in that environment is determined solely by the body of the lambda expression. A compiler for scheme will therefore compile the `lambda` like any other procedure, when it compiles the enclosing procedure. So, for example, when our example procedure make-counter is compiled, the compiler will also compile the code for the `lambda` body. This code will be kept around for use by `make-counter`.

The actual run-time code for `lambda` just consists of fetching the address of the code, and the current environment pointer, and putting them in a closure object on the heap. `lambda` is therefore about as fast as `cons`—all that's really happening is the creation of the closure object itself, not anything expensive like calling the compiler at run-time.

[ cost of lambda calling is a handful of instructions...]

## 5.3 An Interpreter with `let` and `lambda`

In this section, I'll present a new interpreter for a bigger subset of Scheme; it handles all of the essential special forms of Scheme, except for macro definitions. (A macro facility would be easy to add, as well, and would make it easy to implement the remaining special forms by automatic transformation, in terms of the special forms the interpreter "understands" directly.)

The new interpreter is very much like the one from the last chapter, with three important differences:

It implements local binding environments as well as a top-level environment. Evaluating an expression (such as a `let`) may create a new environment, and subexpressions (such as the `let` body) can simply be evaluated in the new environment by recursive calls to `eval`

It allows new procedures to be defined, creating closures. Closures pair environments with code bodies that are interpreted by the interpreter. Calling a closure is much like evaluating a `let`. The arguments are bound in a local environment (like `let` variables), and the body is interpreted in that environment.

We will treat special forms differently, binding special form names in much the same way as normal variable names. This will make the interpreter cleaner and more extensible.

Here is our new `eval`:

```
(define (eval expr envt)
   (cond ((symbol? expr)
          (eval-symbol expr envt))
         ((pair? expr)
          (eval-list expr envt))
         ((self-evaluating? expr)
          expr)
         (#t
          (syntax-error "Illegal expression form" expr))))
```

Notice that not much has changed—`eval` still just analyzes expressions and dispatches to more specialized helper procedures that handle particular kinds of expressions.

The important difference is that `eval` expects and environment argument `envt`, which represents the binding environment in which to evaluate an expression.

## 5.3.1 Nested Environments and Recursive Evaluation

Instead of using the old "flat" representation of an environment, which was just a table of name-value pairs, we'll represent nested environments as a list of tables, or *environment chain.*

When we begin interpretering, the environment chain will consist of one table, the top-level environment. When we evaluate a binding construct such as a `let`, we will create a new table, or *enviornment frame*, which binds the local variables. This frame will contain the name-value pairs bound locally, plus a pointer to the next enclosing environment. The environment chain is thus a linked list that acts like a stack, for the most part—new enviornment frames are pushed on the front of the list when entering a binding construct, and popped off the front of the list when exiting it.

We could implement this stack-like behavior with an explicit stack data structure in the interpreter, but it's easier to use the activation "stack" of the language we're using to implement the interpreter. (In this case, that happens to be Scheme, but if we were implementing the interpreter in C, we could use C's activation stack.)

At any given point during evaluation, the *current environment* is the environment referred to by the interpreter's variable `eval`, an in particular the most recent binding of `eval`.

When we evaluate an expression that doesn't change the interpretive environment, and call `eval` recursively to evaluate subexpressions, we simply pass the `envt` variable's value to the recursive calls. This will ensure that the subexpressions execute in the same environement as the containing expression.

When we evaluate a binding construct, and evaluate subexpressions in that environment, we create a new environment and pass *that* to the recursive calls to `eval`, so the subexpressions will execute in the new enviornment instead.

Notice that we don't actually modify the environment chain when creating a new environment—we simply create a new frame which holds a pointer to the old environment, and pass it to the recursive `eval`. The fact that we don't actually modify the structure of the environment is important—it's will let us implement closure correctly.

When the interpreter returns from evaluating a subexpression, it returns to an enclosing invocation of `eval`; the old environment will become visible again because we return to an `eval` where that environment is the value of the `envt` argument.

For example, consider what happens when we interpret the following expression, starting at the top level

```
(let ((foo 1))
   (if (a)
       (let ((bar 2))
          (if (b)
              (c)
              (d))
          (e))
       (f)
   (g))
```

[ We'll focus on the nested calls to eval corresponding to the nesting of let, if, let, if ]

If we look at the nested calls to `eval`, we first see a call that evaluates the whole expression in the top-level environment:

```
                                    +------+
eval  expr: (let...)  envt: |  *--+--> [toplevel envt]
                                    +------+
```

(I've given a textual representation of the `expr` argument, but a pictorial representatio of the `envt` argument.)

`eval` will dispatch to `eval-let`, passing it the same environment. `eval-let` will evaluate the initial value expression `1` in that environment, and create a new environment binding `foo`. (I'll ignore the recursive call to eval to evaluate the argument.) It will then call `eval` recursively to evaluate the `let` body in that environment.

I'll depict the nested invocations of `eval` and `eval-let` top-to-bottom, showing the stack growing twoard the bottom of the picture. (This just turns out to be simpler than drawing the stack growing up.)

```
                                    +-----+
eval     expr: (let...)   envt: |  *--+--> [toplevel envt]
                                    +-----+      /|\     /|\
                                                  |       |
                                    +-----+       |       |
eval-let expr: (let...)   envt: |  *--+-------+       |
                                    +-----+               |
                                                          |
                                    +-----+               |
eval     expr: (if...)    envt: |  *--+-->  [ [foo 1] * ]
                                    +-----+
```

eval-if will evaluate the condition expression (a) in the given environment. We'll ignore that recursive call to eval, but assume it returns a true value. In that case, eval-if will evaluate its consequent, the inner let expression, by another recursive call to eval.

At this point, the "stack" of invocations of eval, eval-let, and eval-if looks like this:

```
                                    +-----+
eval     expr: (let...)   envt: |  *--+--> [toplevel envt]
                                    +-----+      /|\     /|\
                                                  |       |
                                    +-----+       |       |
eval-let expr: (let...)   envt: |  *--+-------+       |
                                    +-----+               |
                                                          |
                                    +-----+               |
eval     expr: (if...)    envt: |  *--+---> [ [foo 1] * ]
                                    +-----+      /|\
                                                  |
                                                  |
                                    +-----+       |
eval-if  expr: (if...)    envt: |  *--+-------+
                                    +-----+       |
                                                  |
                                    +-----+       |
eval     expr: (let...)   envt: |  *--+-------+
                                    +-----+
```

Again, the `let` will evaluate the intial value expression, `2`, by a recursive call to `eval`, which we will ignore here. Then it will bind `bar` in a new environment frame, and call `eval` recursively to evaluate the body in that environment. The body consists of another if, so `eval-if` will be called, and it will evaluate its argument expression and either the consequent or the alternative in that environment.

Assuming the condition returns true and it evaluates the consequent, `(c)`, here's the "stack" of invocations of `eval`, `eval-let`, and `eval-if` at the point where `(c)` is evaluated:

```
                                +-----+
eval      expr: (let...)   envt: |  *--+--> [toplevel envt]
                                +-----+      /|\      /|\
                                              |        |
                                +-----+       |        |
eval-let expr: (let...)   envt: |  *--+-------+        |
                                +-----+                |
                                                       |
                                +-----+                |
eval      expr: (if...)    envt: |  *--+---> [ [foo 1] * ]
                                +-----+      /|\      /|\
                                              |        |
                                              |        |
                                +-----+       |        |
eval-if  expr: (if...)    envt: |  *--+-------+        |
                                +-----+       |        |
                                              |        |
                                +-----+       |        |
eval      expr: (let...)   envt: |  *--+-------+        |
                                +-----+                |
                                                       |
                                +-----+                |
eval      expr: (if...)    envt: |  *--+---> [ [bar 2] * ]
                                +-----+      /|\
                                              |
                                +-----+       |
eval      expr: (c)        envt: |  *--+-------+
                                +-----+
```

[ Note that the pictures above all depict evaluation of nested *non-tail* expressions. In the case of tail expressions, the "stack" will not include as much information, because the state of the calls to `eval`, etc., will not be saved before the calls that evaluate subexpressions.

Our interpreter is written in good tail-recursive style, with tail calls to evaluate expressions that are tails of expressions in the language we're interpreting. This means that the intepreter is tail-recursive wherever the program it's implementing is tail-recursive, and since it's implemented in a tail-recursive language (Scheme), we preserve the tail-recurson of the program we're interpreting. In effect, we snarf tail-call optimization from the underlying Scheme system. If we were implementing our interpreter in C, we'd have to use special tricks to preserve tail recursion. We'll show how this can be done later, when we discuss our compiler. ]

### 5.3.2 Integrated, Extensible Treatment of Special Forms

In the interpreter in the last chapter, we implemented special forms directly in the interpreter—`eval-list` checked compound expressions to see if they began with a special form name. In effect, we hardcoded the meanings of special form names in the procedure `eval-special-form`.

In our new interpreter, we'll use a cleaner approach, which treats special form definitions pretty much like variable definitions. This will let us put special forms in particular environments, and use the normal scoping mechanisms to look up the routines that compile them.

This has several advantages. The first is that it makes our interpreter more modular. We can create different environments with different special forms, and use the same interpreter to interpret different languages. That is, we separate out the basic operation of the interpreter from the particular special forms we decide on.

The second advantage is that it will allow us to build an elegant macro facility, so that new special forms can be defined in terms of old ones. (This will be described in detail in [ a later chapter ].)

[ this is out of place, but fwd ref idea anyway? Shorten? Or just move?]

A Scheme interpreter or compiler only needs to "understand" procedure calling and a few basic special forms—`if`, `lambda`, `set!`, `quote`, [did I leave one out?], and one very special special form for defining new special forms (macros). (We can write `cond` as a macro using `if`, `let` as a macro using `lambda`, `letrec` as a macro using `let`, `lambda`, and `set!`, and so on.)

The third advantage is that we can use the same scoping rules for special forms that we use for variables. This will be very convenient later, because we will be able to define local macros, in much the same way we define local procedures.

To support this, we need to represent bindings slighly differently. In the simple interpreter from the last chapter, each binding was just a name-value pair. Now we'll have a third part to each binding, telling what kind of binding it is—a variable binding, a special form binding, or a macro binding.

We can still use associations to represent the bindings. Where the simpler interpreter representing each binding as an association of the form `(name value)`, the new one will use bindings of the form `(name type whatever)`. In the case of a normal variable binding, the "whatever" is the actual value of the variable. In the case of a special form, the "whatever" is the information the interpreter needs to interpret that particular special form, including the procedure to evaluate it. For example, when binding the name `let`, we can store a pointer to the procedure `eval-let` right there in the binding information.

Since the exact representation of bindings is irrelevant, and we may want to change it, we'll call the whole thing a `binding-info` data structure. This reflects that fact that it may not hold just a binding, but also any auxiliary information we want to store.

To abstract away from exactly how bindings are implemented, we'll define several procedures that operate on `binding-info`'s. These include:

`binding-type`, which returns a symbol saying what kind of binding it is: `<variable>` for a normal variable, `<special-form>` for a built-in special form binding, and `<syntax>` for a syntax (macro) binding.

`bdg-variable-ref`, which returns the value of a normal variable binding.

`bdg-special-form-evaluator`, which returns an evaluation procedure for a special form binding.

For now we'll ignore `<syntax>` bindings, which will be discussed in a later chapter.

[ give actual code for accessors, etc? ]

Here's our new `eval-list` for handling compound expressions:

```
(define (eval-list list-expr envt)
   ; only try to consider it specially if the head is a symbol
   (if (symbol? (car list-expr))

       ; look it up in the current lexical environment
       (let ((binding-info (envt-lexical-lookup envt (car list-expr))))

          ; switch on the type of thing that it is
          (cond ((not binding-info)
                 (run-time-error "Unbound symbol" (car list-expr)))
                (else
                 (cond
                    ; special forms just call the special-form
                    ; evaluator, which is stored in the binding-info
                    ; object itself
                   ((eq? (binding-type binding-info) '<special-form>)
                    ((bdg-special-form-evaluator binding-info) list-expr envt))

                   ((eq? (binding-type binding-info) '<variable>)
                    (eval-combo (bdg-variable-ref binding-info)
                                (cdr list-expr)
                                envt))
                   ((eq? (binding-type binding-info) '<syntax>)
                    (eval-macro-call (bdg-syntax-transformer binding-info)
                                     list-expr
                                     envt))
                   (else
                    (internal-error "Unrecognized binding type"))))))

         ; the head of the list is not a symbol, so evalute it
         ; and then do an eval-combo to evaluate the args and
         ; call the procedure
         (eval-combo (eval (car list-expr) envt)
                     (cdr list-expr)
                     envt)))
```

`eval-list` first checks to see whether the head of the list is a symbol; if not, it's just a combination (procedure call expression), and is handled by `eval-combo`. (Remember that a combination can have an arbitrary expression as its operator, and that expression is assumed to return a procedure to call.)

If it is a symbol, the binding of the variable is looked up. If it's a special from binding, the evaluation procedure is extracted from the binding info, and called to evaluate the expression.

If the head of the list is just the name of a normal variable, that's also just a combination, and `eval-combo` is called in that case, too.

If the head of the list is the name of a syntax binding (macro), we call `eval-macro-call` to deal with it; don't worry about this for now—it will be discussed in detail in Chapter [ whatever ].

Notice that in all cases, the environment is passed along unchanged to whatever procedure handles the expression.

### 5.3.3 Interpreting `let`

The procedure `eval-let` will be stored in the binding for the special form `let`. In the case of a `let` expression, `eval-let` (above) will extract this procedure from the binding and call it to evaluate the expression.

```
(define (eval-let let-form envt)

   ; extract the relevant portions of the let form
   (let ((binding-forms (cadr let-form))
         (body-forms (cddr let-form)))

      ; break up the bindings part of the form
      (let ((var-list (map car binding-forms))
            (init-expr-list (map cadr binding-forms)))

         ; evaluate initial value expressions in old envt, create a
         ; new envt to bind values,
         (let ((new-envt (make-envt var-list
                                    (eval-multi init-expr-list envt)
                                    envt)))
            ; evaluate the body in new envt
            (eval-sequence body-forms new-envt)))))
```

The first thing `let` does is to extract the list of variable binding clauses and the list of body expressions from the overall `let` expression. Then it further decomposes the variable binding clauses, extracting a list of names and a corresponding list of initial value expressions. (Notice how easy this is using `map` to create lists of `car`'s and `cadr`'s of the original clause list.)

`eval-let` then calls a helper procedure, `eval-multi`, to recursively evaluate the list of initial value expressions and return a list of the actual values.

Then it calls `make-envt` to make the new environment. This creates a new environment frame, scoped inside the old environment—i.e., with a scope link to it—with variable bindings for each of the variables, initialized with the corresponding values.

Then `eval-let` calls `eval-sequence` to recursively evaluate the body expressions in the new environment, in sequential order, and return the value of the last expression. This value is returned from `eval-let` as the value of the `let` expression.

Here's the code for `eval-multi`, which just uses `map` to evaluate each expression and accumulate a list of results.

`eval-multi` calls `eval` recursively to evaluate each subexpression in the given environment. To do this, it must pass `two` arguments to `eval`. It uses `map` to iterate over the list of expressions, but instead of calling `eval` directly, `map` calls a helper procedure that takes an expression as its argument, and then passes the expression *and* the environment to `eval`.

Recall from section [ whatever ] that technique is known as *currying*. We use `lambda` to create a specialized version of a procedure (in this case `eval`), which automatically supplies one of the arguments. In effect, we create a specialized, one-argument version of `eval` that evaluates expressions in a particular environment, and then map that procedure over the list of expressions.

```
(define (eval-multi arg-forms envt)
   (map (lambda (x)
           (eval x envt))
        arg-forms))
```

Here's the code for `eval-sequence`, which is very much like `eval-multi`—it just evaluates a list of expressions in a given environment. It's different from `eval-multi` in that it returns only the value of the last expression in the list, rather than a list of all of the values.

```
(define (eval-sequence arg-forms envt)
   (if (pair? arg-forms)
       (cond ((pair? (cdr arg-forms))
              (eval (car arg-forms) envt)
              (eval-sequence (cdr arg-forms) envt))
             (else
              (eval (car arg-forms) envt)))
       '*undefined-value*)) ; the value of an empty sequence
```

(Notice that we've written `eval-sequence` tail-recursively, and we've been careful to evaluate the last expression using a tail-call to `eval`. This ensures that we won't have to return to `eval-sequence`, so if the expression we're interpreting is a tail-call, we won't lose tail-recursiveness in the interpreter.)

### 5.3.4 Variable References and `set!`

`eval-symbol` handles variable references. It looks up the binding of the symbol, if there is one—if not, it's an unbound variable error—and checks to see that it's a variable reference and not a special form or macro. If it is a normal variable, it fetches the value from the binding and returns it.

```
(define (eval-symbol name-symbol envt)
   (let ((binding-info (envt-lexical-lookup envt name-symbol)))
      (cond ((not binding-info)
              (run-time-error "Symbol does not have value"
                              name-symbol))
             ((eq? (binding-type binding-info) '<variable>)
              (bdg-variable-ref binding info))
             (else
              (syntax-error "non-variable name referenced as variable"
                             name-symbol)))))
```

`eval-set!` handles the `set!` special form. It will be stored in a special form binding of the name `set!`, and extracted and called (by `eval-list`) to evaluate `set!` expressions.

```
(define (eval-set! set-form envt)
   (let ((name (cadr set-form))
         (value-expr (caddr set-form)))
      (let ((binding-info (envt-lexical-lookup envt name)))
         (cond ((not binding-info)
                 (run-time-error "Attempt to set! undeclared variable" name))
                ((eq? (binding-type binding-info) '<variable>)
                 (bdg-variable-set! binding-info (eval value-expr envt)))
                (else
                 (syntax-error "Attempt to set! a non-variable" name))))))
```

### 5.3.5 Interpreting `lambda` and Procedure Calling

Our new interpreter will handle defining and calling new procedures. This is not difficult, because all of the major mechanisms are already in place. We need the ability to define local variables (e.g., arguments), which we already implemented for `let`. We also need the ability to interpret the procedure bodies, but the interpreter we've got is just fine for that. We'll simply store the procedure bodies as s-expressions, and interpret them like any other expressions when the procedure is called.

Our representation of closures will be very simple. A closure mainly pairs an environment with a procedure body, but we also need to specify a list of argument the procedure will accept.

We'll define a procedure `make-closure` to construct a closure, given a pointer to an environment, a pointer to a list of argument names (symbols), and pointer to a procedure body (a list of expressions).

We'll also define the procedures `closure-envt`, `closure-args`, and `closure-body` to extract those parts when we call the procedure.

As a slight complication, we'd like to start out with some predefined procedures, and the easiest way to do that is simply to snarf the corresponding procedures from the underlying Scheme system, i.e., the language we're using to implement our interpreter. (If we were writing our interpreter in C or assembly language, we might write the code bodies of built-in procedures in that language.)

These snarfed procedures will be the built-in "primitive" operations in our language, which can be "glued together" by the interpreter to build new procedures, which may be arbitrarily complicated.

In the simple interpreter in the last chapter, we snarfed procedures directly—we just used closures in the underlying Scheme as procedures in our language. In the new interpreter, we need to distinguish between snarfed procedures (which we can simply call from inside the interpreter) and user-defined procedures, which we must interpret via recursive calls to `eval`.

Our representation of closures will therefore support two predicates. `closure?` will test an object to see if it is a closure of either sort. `primitive-closure?` will test whether a closure represents a snarfed procedure from the underlying Scheme system.

In the case of a primitive closure, calling the closure just consists of extracting the underlying Scheme closure, and calling it with the given argument values. (We don't snarf any procedures that depend on what environment they execute in. We only snarf functions like `+` and `cons`, which depend only on their arguments.)

A closure therefore has three important fields: a pointer to an environment, a pointer to a list of argument names, and a pointer to a code body. It also has a "hidden" type field, saying that what kind of object it is.

[ I'm glossing over the actual representation in the underlying Scheme system, because it really doesn't matter. It could be an association list, a vector, or whatever. ]

`eval-lambda` is the procedure called from `eval-list` to handle `lambda` expressions. It will be stored in binding of `lambda` of the name `lambda` (with binding type `<special-form>`, and extracted and called to actually interpret `lambda`'s.

```
(define (eval-lambda lambda-form envt)
   (let ((formals (cadr lambda-form))
         (body (cddr lambda-form)))
      (make-closure envt formals body)))
```

`eval-lambda` simply extracts the argument list and body expression list from the `lambda` expression, and calls `make-closure` with them (and the current environment) to create the closure object. Storing the current environment in the closure ensures that when the closure is interpreted later, it will still be able to refer to the same bindings that were visible when it was created.

`eval-combo` is called from `eval-list` to evaluation combinations (procedure call expressions).

(Note that `eval-list` evaluates the operator expression before calling `eval-combo`, and hands it the closure plus a list of unevaluated argument expressions. This is not particularly significant—we could have passed the operator expression to `eval-combo` unevaluated, like the argument expressions, and have `eval-combo` evaluate it instead. As we've written it, we ensure that the operator expression is evaluated before the arguments. We could change it to get the opposite effect. This would still be legal—the Scheme standard does not specify the order of evaluation, and an implementation may even use different orders at different call sites.)

[ DONOVAN—maybe we should change it. RScheme evaluates the operator expression last, so maybe the interpreter should, too. ]

`eval-combo` evaluates the argument expressions in the given environment to get the argument values, using `eval-multi`, and calls `eval-apply` to call the given closure with those values.

```
(define (eval-combo proc arg-expr-list envt)
   ; use our own kind of apply to run our own kind of closures
   (eval-apply proc
               ; evaluate the arguments, collecting the results into a list
               (eval-multi arg-expr-list
                           envt)))
```

`eval-apply` does the actual procedure call, after the arguments have been evaluated. That is, it *applies* the given procedure (closure) to the given arguments.

If the closure we're calling is a primitive closure, we simply extract the underlying Scheme procedure and call that, using the standard Scheme procedure `apply`. Scheme's `apply` takes a list of any number of values, and calls the procedure as though the arguments had been passed to it in the normal way.

(To make sure that you understand that, here's a simple usage of Scheme's `apply`: `(apply + '(1 2))`. This call to apply will take the procedure `+` and call it with the values `1` and `2`, just as if we had written `(+ 1 2)`. Likewise, `(apply list '(1 2 3 4))` returns the same thing as `(list 1 2 3 4)`.)

```
(define (eval-apply proc arg-list)
   (if (primitive-closure? proc)

       ; it's a primitive, so extract the underlying language's
       ; closure for the primitive, and do a real (underlying Scheme)
       ; apply to call it
       (apply (closure-primitive proc) arg-list)

       ; it's not a primitive closure, so it must be something
       ; we created with make-closure
       ;
       ; first, bind the actuals into a new environment, which
       ; is scoped inside the environment in which the closure
       ; was closed
       (let ((new-envt (make-envt (closure-args proc)
                                  arg-list
                                  (closure-envt proc))))
          ; then, evaluate the body forms, returning the
          ; value of the last of them.
          (eval-sequence (closure-body proc)
                         new-envt))))
```

In the case of a user-defined (interpreted) closure, `eval-combo` creates a new environment to bind the arguments values, much as it does to bind the local variables of a `let`; it calls `make-envt` with the name list, the corresponding value list, and the old environment, and gets back a pointer to the new environment frame, scoped inside the old one.

There's a big difference here, though. The "old" environment that's used in creating the new one is *not* the environment that was passed to *eval-combo*. (Notice that *eval-combo* did not even pass that environment to `eval-apply`.)

When we call the closure, we extract the environment stored in the closure, and use that as the "old" environment. This ensures that the closure body will evaluate in the environment where it was defined, augmented with the bindings of its arguments. This is the crucial step in preserving lexical scope—the meanings of identifiers in the procedure body are fixed at the moment the closure is created, because it captures the current environment at that point.

Once the new environment is created, `eval-combo` simply calls `eval-sequence` to evaluate the sequence of body expressions and return the value of the last one. `eval-combo` simply returns this value as the return value of the procedure call. (Notice that the call to `eval-sequence` is a tail call, preserving the tail recursion of the program we're interpreting.)

#### 5.3.5.1 Mutual Recursion Between Eval and Apply

It is important to understand the relationship between `eval` and `apply` in the interpreter. This will help you understand how scoping is implemented, and will also help you understand the relationship between an interpreter and a compiler.

`eval` calls *itself* to evaluate normal nested expressions. It may do this indirectly, by using helper procedures that discriminate different kinds of expressions, but in general recursive calls to eval correspond to the nested structure of a procedure.

`apply` is very different. When the interpreter gets to a procedure call, it calls `apply` to jump to a *different* procedure, not a nested expression of the same procedure. (Note that the arguments to a procedure call are evaluated like any other nested expressions, by calling `eval`, but the call itself is done by `apply`.)

Normal recursive calls to `eval` therefore correspond to the local nesting structure of the code, but calls to `apply` correspond to transfers of control to different procedures.

[ Any other miscellaneous stuff I should explain? Should have a pointer to the source file for the whole interpreter... ]

[ Say that's it for the interpreter for now... we'll come back to it when we talk about macros, and we'll talk about a compiler with very similar structure later... ]

## 5.4 Variants of `let`: `letrec` and `let*`

Scheme provides two useful variants of `let`. `letrec` supports the creation of recursive local procedures, including mutually recursive sets of procedures. `let*` supports the sequenced binding of variables, where each initial value expression can use the previous bindings.

### 5.4.1 Understanding `letrec`

When a normal `let` is evaluated, the initial value expressions are evaluated *before* binding is done. The initial value expressions execute in the environment outside the let, and then the bindings are created and initialized with those values.

Often, we want the initial value expression for a binding to be able to create a procedure that will see the new bindings. For example, suppose we want to create a local procedure which is recursive. We might try this:

```
; buggy example with tail-recursive local procedure
(define (some-procedure...)
   (let ((helper (lambda (x)
                   ...
                   (if some-test?
                       (helper ...))))) ; broken recursive call
      ...
      (helper ...)  ; call to recursive local procedure
      ...))
```

The problem with this example is that when the `let` is evaluated, the `lambda` expression will create the helper procedure in the wrong environment—before the variable `helper` is bound. The resulting procedure will be scoped in the environment outside the `let`, not the new environment where `helper` is visible. When the procedure calls `helper`—which we had intended to be a recursive call—it will not use new binding of `helper` that we created. Inside the `lambda` body, `helper` will still refer to whatever binding of `helper` was visible before intering the let. (Very likely, that's no variable at all, and this will cause an unbound variable error.)

`letrec` lets us create an environment *before* evaluating the initial value expressions, so that the initial value computions execute inside the new environment. We can fix the problem by using a `letrec` instead of a `let`:

```
(define (some-procedure...)
   (letrec ((helper (lambda (x)
                      ...
                      (if some-test?
                          (helper ...))))) ; recursive call
      ...
      (helper ...)  ; call to recursive local procedure
      ...))
```

Now the procedure `helper` can "see its own name," since the lambda expression is evaluated in the environment where `helper` is bound.

A `letrec` expression is equivalent to a `let` where the bindings are initialized with dummy values, and then the initial values are computed and assigned into the bindings. The above example is equivalent to:

```
(define (some-procedure ...)
   (let ((helper '*dummy-value*))
      (set! helper (lambda (x)
                      ...
                      (if some-test?
                          (helper ...))))) ; recursive call
     ...
     (helper ...)  ; call to recursive local procedure
     ...))
```

`letrec` can be used when defining mutually recursive procedures, each of which can see the others' names and call them.

```
(define (some procedure ...)
   (letrec ((helper1 (lambda ()
                        ... (helper2) ...))
            (helper2 (lambda ()
                        ... (helper1) ...)))
     ...
     (helper1)  ; start up mutual recursion
     ...))
```

Notice that all `letrec` does is bind variables and (re-)initialize them. You can use it to define plain variables as well as procedure variables. For example, if the recursive procedures above need to reference a shared variable, you can do this:

```
(define (some procedure ...)
   (letrec ((helper1 (lambda ()
                        ... var1 ... (helper2) ...))
            (helper2 (lambda ()
                        ... (helper1) ... var1 ...))
            (var1 #f))
     ...
     (helper1)  ; start up mutual recursion
     ...))
```

[ should come up with some simple concrete examples...]

As with `let`, the order of evaluation of a `letrec`'s initial value expressions is undefined. For example, the above `letrec` might be compiled as though it were a `let` like this:

```
(define (some procedure ...)
   (let ((helper1 '*dummy-value*)
         (helper2 '*dummy-value*)
         (var1 '*dummy-value*))
      (set! helper2 (lambda ()
                        ... (helper1) ... var1 ...))

      (set! var1 #f)
      (set! helper1 (lambda ()
                       ... var1 ... (helper2) ...))
     ...
     (helper1)  ; start up mutual recursion
     ...))
```

When using `letrec` and `lambda` to define local procedures, in the usual way, the order of evaluation is irrelevant—the `lambda` expressions can be executed in any order, because they only refer to the `bindings` of the `letrec` variables, not their `values`. The values are only used when the resulting procedures are called. The following would be an error, however:

```
(define (some procedure ...)
   (letrec ((helper1 ...)
            (helper2 ...)
            (var1 (list helper1 helper2)))
      ...
      ((car (var1 helper1)))  ; start up mutual recursion
      ...))
```

Here the initialization of `var1` depends on the values of `helper1` and `helper2`, which may not have been computed yet.

#### 5.4.1.1 Using `letrec` and `lambda` to Implement Modules

Standard Scheme does not have a module system, but `letrec` and `lambda` are powerful enough to implement modules in portable Scheme.

Suppose we would like to define a module that encapsulates a set of procedures and variables, but only exports a subset of those procedures.

We can represent the module as a `letrec` environment which exports an association list of of procedures.

Here we'll create a module called `foo` module, which defines four procedures and two variables, and exports two of the procedures, `foo` and `bar`.

```
(define foo-module
   ; create a letrec environment with internal definitions
   ; of some variables and procedures
   (letrec ((private-proc1 (lambda (...) ...))
            (private-proc2 (lambda (...) ...))
            (private-var1 ...)
            (private-var2 ...)
            (foo (lambda (...) ...))
            (bar (lambda (...) ...)))
      ; return an association list of "exported" closures
      (list (list 'foo foo)
            (list 'bar bar))))
```

The letrec expression will create an environment, and within that environment it will evaluate the initial value expressions to initialize the bindings. All of the procedures in the letrec can see each other's names, and call each other freely. Procedures outside the letrec cannot.

The only procedures that can be called from outside the letrec are foo and bar, which are returned from the letrec in an association list. We've saved this list in the binding of foo-module, so that we can look those procedures up and call them.

We can clean this up a little by providing an accessor function that will extract a single procedure from a module, by using assq to find the appropriate closure:

```
(define (module-get mod name)
   (cadr (assq mod name)))
```

To import a procedure and give it a name in another environment, we can do this:

```
(define foo (module-get foo-module 'foo))
```

If we want to, we can give it a different name in the environment we're "importing" it into.

```
(define quux (module-get foo-module 'foo))
```

This lets us rename a procedure imported from a module, to avoid naming conflicts. quux is exactly the same procedure as foo, but by a different name in a different scope. When we call it, it will execute in the environment where it was defined, namely the "private" environment of the module we created with letrec.

### 5.4.2 let*

For situations where the order of initialization is important, Scheme provides a variant of let called let*.

Suppose we tried the following using let:

```
(define (foo x)
   (let ((a 0)
         (upper (+ a epsilon))
         (lower (- a epsilon)))
     ...))
```

This will not do what we probably meant, because the initial values of `upper` and `lower` will be computed before `a` is bound. We could fix this by using nested `let`'s, to force evaluation and binding to happen in the desired order:

```
(define (foo epsilon)
   (let ((a 0))
      (let ((lower (- a epsilon))
            (upper (+ a epsilon)))
     ...))
```

This ensures that `a` is bound before we evaluate the initial value expressions for `upper` and `lower`.

Scheme provides `let*` to avoid needing lots of nested lets when initilizing a series of bindings, each of which may depend ont the previous ones, e.g.,

```
(define (bar x y)
   (let* ((diff (- x y))
          (diff-squared (* diff diff))
          (diff-cubed (* diff-squared diff)))
     ...)
```

is exactly equivalent to

```
(define (bar x y)
   (let ((diff (- x y)))
      (let ((diff-squared (* diff diff)))
         (let ((diff-cubed (* diff-squared diff)))
            ...))))
```

## 5.5 Iteration Constructs

### 5.5.1 Named `let`

Named `let` is a general and flexible iteration construct that is really just syntactic sugar for a `letrec` and one or more `lambda`'s. It looks like a `let`, but it's usually used as a loop.

Named `let` implements iteration as recursion. If you use it in normal ways, you write loops that act as tail-recursive procedures. You can also use it to write "loops" that aren't tail recursive, but that's uncommon.

Named `let` binds loop variables, and executes the loop body. Anywhere in the loop body, you can call a procedure to iterate the loop.

Here's an example loop, which prints out the integers from 0 to 9:

```
(let loop ((i 0))
   (display i)
   (if (< i 10)
       (loop (+ i 1))))
```

Here we've written a loop and given it an identifier, `loop`; that's just a name we chose for this particular loop—we could have used any identifier.

This loop binds the loop variable `i`, giving it the intial value 0. Then it enters the body of the loop, which prints out `i` using display, and evaluates the `if` expression. If the `if` condition returns a true value, it evaluates the expression `(loop (+ i 1))`, which iterates the loop. This looks like a call to a procedure named `loop`, which iterates the loop. The argument passed is the new value of the loop variable for the next iteration.

The reason that the expression that iterates a loop looks like a procedure call is that it *is* a procedure call. A named `let` is exactly equivalent to a `letrec` that defines a named procedure, whose body is the body of the named `let`, and then calls that procedure to start the recursion. When you write a "loop" with named `let`, you're really writing a recursive procedure and a call to that procedure. The loop variable(s) are really arguments to the procedure, and the initial values of the loop variables are just the first argument passed to the procedure to start the recursion.

The above example is exactly equivalent to:

```
(letrec ((loop (lambda (i)      ; define a recursive
                  (display i)   ; procedure whose body
                  (if (< i 10)  ; is the loop body
                      (loop (+ i 1))))))
   (loop 0)) ; start the recursion with 0 as arg i
```

When you supply the name of a named `let`, you're really supplying the name of a `letrec` variable that will name a procedure. When you supply the body of the named `let`, you're really supplying the body of the named procedure. When it iterates the loop, it is calling itself recursively, passing the new invocation the new value of the loop variable as an argument.

To start off the loop, named `let` passes this procedure the initial value expression for the loop variable.

We can provide any expression we want to compute the new value of the loop variable—we don't have to increment it by one. We can also provide any test we want to decide whether to iterate the loop.

For example, here's procedure which uses a loop to search a list of alternating key/value pairs. (This is not an association list, but a linear list of alternating keys and values, called a property list.) It iterates through the list two elements at a time. If it finds an odd-numbered element that's eq? to what it's looking for, it returns the next (even-numbered) element; otherwise, it continues through the loop.

```
(define (property-list-search lis target)
   (let loop ((l lis))
      (cond ((null? l)
              #f)
             ((eq? (car l) target)
              (cadr l))
             (#t
              (loop (cddr l))))))
```

[ same as: ]

```
(define (property-list-search lis target)
   (letrec ((loop (lambda (l)
                    (cond ((null? l)
                           #f
                          ((eq? (car l) target)
                           (cadr l))
                          (#t
                           (loop (cddr l))))))))
      (loop lis)))   ; start the recursion
```

The reason we supply a name for a loop in a named `let` is so that we can have nested loops with different names, and we can iterate any of the loops by calling it by name.

For example, suppose we want to have a nested pair of loops, but want to be able to bail out of the iteration of the inner loop, and go directly to the next iteration of the outer loop. We can do this:

[ need example here ]

```
(let outer-loop ((i ...))
   (let inner-loop ((j ...))
      (if (should-abort-inner-loop)
          (outer-loop ...)) ; go directly to next iteration of outer loop
      ...  ; do normal inner loop action
      ...(inner-loop ...)  ; iterate inner loop normally
   ...
   ... (outer-loop ...))   ; iterate outer loop normally
```

Some things to notice about Scheme loops:

Loops can have any number of loop variables, each updated in any way you like. This corresponds to having a recursive procedure with any number of arguments, and passing it any values you like at each recursion.

Unlike most languages' loops, each time we iterate a loop, we *rebind* the loop variable. There's a new binding at each iteration, because each iteration is really a call to a procedure that binds arguments. We don't bind the loop variable once and side-effect it at each iteration.

Since loop bodies are really just procedure bodies, and loop iterations are really just procedure calls, we can put calls that iterate a loop anywhere in the body; we can have multiple points in the body that call the procedure to iterate the loop.

The variable bindings created at each iteration of a loop are independent, and can be captured by lambda expressions in the loop body. Each closure created by lambda will capture the bindings for *that* iteration of the loop.

## 5.6 Programming with Procedures and Environments

## 5.7 `do`

Scheme provides an iteration construct, `do` similar to `do` or `for` loops in other languages.

[ blah blah ... ]

## 5.8 Exercises

# 6 Recursion in Scheme

In this chapter, I'll discuss procedure calling and recursion in more depth. [ blah blah blah ] Scheme's procedure-calling mechanism supports efficient tail-recursive programming, where recursion is used instead of iteration.

After clarifying how recursion works, I'll give examples of how to program recursively in Scheme.

(In a later chapter, I'll show how the mechanisms that support tail recursion also support a powerful control feature called `call-with-current-continuation` that lets you implement novel control structures like backtracking and coroutines.)

## 6.1 Subproblems and Reductions (non-tail and tail calls)

In most implementations of most programming languages, an activation stack is used to implement procedure calling. At a call, the state of the "caller" (calling procedure) is saved on the stack, and then control is transferred to the callee.

Because each procedure call requires saving state on the stack, recursion is limited by the stack depth. In many systems, deep recursions cause stack overflow and program crashes, or use up unnecessary virtual memory swap space. In most systems, recursion is unnecessarily expensive in space and/or time. This limits the usefulness of recursion.

In Scheme, things are somewhat different. As I noted earlier, recursive calls may be *tail recursive*, in which case the state of the caller needn't be saved before calling the callee.

More generally, whether a procedure is recursive or not, the calls it makes can be classified as *subproblems* or `reductions` If the last thing a procedure does is to call another procedure, that's known as a reduction—the work being done by the caller is complete, because it "reduces to" the work being done by the callee.

For example, consider the following procedures:

```
(define (foo)
   (bar)
   (baz))
```

```
(define (baz)
   (bar)
   (foo))
```

Notice that when `foo` is called, it does two things: it calls `bar` and then calls `baz`. After the call to `bar`, control must return to `foo`, so that it can continue and call `baz`. The call to `bar` is therefore a subproblem—a step in the overall plan of executing `foo`. When `foo` calls `baz`, however, that's *all* it needs to do—all of its other work is done.

In a normal programming language implementation, `foo`'s state would be saved before the call to `baz`, as well as before the call to `bar`. Each call would return control to `foo`. In the case of the call to `baz`, all `foo` will do is return the result of the call to *its* caller. That is, all `foo` does after the return from `baz` is to leave the result wherever its caller expects it, and return again to pop a stack frame off the activation stack.

In Scheme, things are actually simpler. If the last thing a procedure does is to call another procedure, the caller doesn't save its own state on the stack. When the callee returns, it will return to its *caller's* caller directly, rather than to its caller. After all, there's no reason to return to the caller if all the caller is going to do is pass the return value along to *its* caller.

In effect, this optimizes away the unnecessary state saving and returning at tail calls.

Consider both `foo` and `baz` above. Neither ever returns—each just calls the other. In Scheme, these two procedures will repeatedly call each other, without saving their state on the stack, producing an infinite mutual recursion. Will the stack overflow? No. Each will save its state before calling `bar`, but the return from `bar` will pop that information off of the stack. The infinite tail-calling beetween `foo` and `baz` will not increase the stack height at all.

Above I said that a callee may return to its caller's caller, but that doesn't really capture the extent of what's going on. In general a procedure may return to its caller (if it was non-tail called), or it's caller's caller (if it was tail-called but its caller wasn't) or it's caller's caller's caller (if it and it's caller were both tail-called), and so on. A procedure returns to the last caller that did a non-tail call.

Because of this "tail call optimization," you can use recursion very freely in Scheme, which is a good thing—many problems have a natural recursive structure, and recursion is the easiest way to solve them.

Notice that this tail call optimization is a feature of the language, not just some implementations—any implementation of standard Scheme is required to support it, so that you can count on it and write portable programs that rely on it.

Also notice that the interpreter we presented earlier is tail-recursive. The recursive calls to `eval` are tail calls, and since it's implemented in Scheme, the interpreter relies on the underlying Scheme's tail-call optimization. The evaluator thus snarfs the tail-call optimization from the underlying Scheme system. If you implement a Scheme interpreter in another language, you have to be more careful, and implement the tail call optimization yourself. This is not actually difficult, as I'll show in the next section.

It's something of a misnomer to call Scheme's procedure calling mechanism an "optimization." What's really going on is that Scheme simply distinguishes between two things that most languages lump together—saving the caller's state, and actually calling the callee. Scheme notices that these things are distinct, and doesn't bother to do the former when only the latter is necessary.

A procedure call is really rather like a (safe) goto that can pass arguments: control is transferred directly to the callee, and the caller has the option of saving its state beforehand. (This is safer than unrestricted goto's, because when a procedure does return, it returns to the right ancestor in the dynamic calling pattern, just as though it had done a whole bunch of returns to get there.)

## 6.2 The Continuation Chain

In this section, I'll describe a straightforward implementation of Scheme's state-saving for procedure calling. Readers uninterested in implementation issues may want to skip it. It may clarify things that are discussed later, however, such as `call-with-current-continuation` and our example compiler's code generation strategy.

In most conventional language implementations, as noted above, calling a procedure allocates an activation record (or "stack frame") that holds the return address for the call *and* the variable bindings for the procedure being called. The stack is a contiguous area of memory, and pushing an activation record onto the stack is done by incrementing a pointer into this contiguous area by the size of the stack frame. Removing a stack frame is done by decrementing the pointer by the size of the stack frame.

Scheme implementations are quite different. As we've explained previously, variable bindings are *not* allocated in a stack, but instead in environment frames on the garbage-collected heap. This is necessary so that closures can have indefinite extent, and can count on the environments they use living as long as is necessary. The garbage collector will eventually reclaim the space for variable bindings in frames that aren't captured by closures.

(Actually, I'm oversimplifying a bit here. Some implementations of Scheme do use a relatively conventional stack, often so that they can compile Scheme straightforwardly to C. They must provide tail-call optimization somehow, though. I won't go into alternative implementation strategies here.)

Scheme implementations also differ from conventional language implementations in how they represent the saved state of callers. (In a conventional language implementation, the callers' state is in two places: the variable bindings are in the callers' own stack frames, and the return address is stored in the callee's stack frame.)

In a Scheme implementation, the caller's state is saved in a record on the garbage-collected heap, called a *partial continuation*. It's called a continuation because it says how to resume the caller when we return into it—i.e., how to continue the computation when control returns. It's called a *partial* continuation because that record, by itself, it only tells us how to resume the caller, not the caller's caller or the caller's caller's caller. On the other hand, each partial continuation holds a pointer to the partial continuation for *its* caller, so a chain of continuations represents how to continue the whole computation: how to resume the caller, and when it returns, how to resume *its* caller, and so on until the computation is complete. This chain is therefore called a *full continuation*.

In most Scheme implementations, a special register called the *continuation register* is used to hold the pointer to the partial continuation for the caller of the currently-executing procedure. When we call a procedure, we can package up the state of the caller as a record on the heap (a partial continuation), and push that partial continuation onto the chain of continuations hanging off the continuation register.

```
                     part. cont.  (saved state of caller's
                         /|\       caller's caller)
                          |
                          |
                     part. cont.  (saved state of caller's caller)
                         /|\
                          |
       +-------+          |
  CONT |   +---+-----> part. cont.  (saved state of caller)
       +-------+
```

(It is often convenient to draw stacks and continuations as growing downward, which is our convention here—the newer elements are on the bottom.)

Note that the continuation register may be a register in the CPU, or it may just be a particular memory location that our implementation uses for this purpose. The point is just that when we're executing a procedure, we always know where to find a pointer to the partial continuation that lets us resume its caller. We will sometimes abbreviate this register's name as `CONT`. A typical implementation of Scheme using a compiler has several important registers that encode the state of the currently-executing procedure:

The *environment register* (`ENVT`) holds the pointer to the chain of environment frames that make up the environment that the procedure is executing in.

The *program counter* register (`PC`) holds the pointer to the next instruction to execute. In a normal system that compiles to normal machine code, this is the actual program counter of the underlying hardware.

The *continuation* register (`CONT`), as we've said, holds the pointer to the chain of partial continuations that lets us resume callers. This is very roughly the equivalent of an activation stack pointer.

Before we call a procedure, we must save a continuation if we want to resume the current procedure after the callee returns.

Since the important state of the currently-executing procedure is in the registers listed above, we will create a record that has fields to hold them, and push that on the continuation chain. We will save the value of the CONT, ENVT, and PC registers in the partial continuation, then put a pointer to this new partial continuation in the continuation registers. We also need to save any other state that the caller will need when it resumes, as suggested by the ellipsis below. (We'll discuss what else goes in a partial continuation when we talk about compilers in detail.)

```
                                   old cont.
                                     /|\
                                      |
                    +-------+         |
     +-------+      |p.cont.|         |
CONT |   +---+------>+=======+        |
     +-------+  cont |   +---+--------+
                    +-------+
               envt |   +---+-------->old envt
                    +-------+
                 pc |   +---+-------->return address
                    +-------+
                            |
                    +  ...
                    |       |
                    +-------+
```

Notice that since we saved the old value of the continuation register in the partial continuation, that serves as the "next" pointer in the linked list that makes up the full continuation. This is exactly as it should be. The value of the continuation register is part of the caller's state, and saving it naturally constructs a linked list, because each procedure's state is fundamentally linked to the state of its caller. Saving the return address is a little bit special—rather than just copying the program counter and saving it, we must save the address we want to resume at when we resume this procedure.

Once a procedure has pushed a continuation, it has saved its state and can call another procedure. The other procedure can use the `ENVT`, `CONT`, and `PC` registers without destroying the values of those registers needed by the caller. This is called a *caller saves* register usage convention; the assumption is that the callee is allowed to freely clobber the values in the registers, so it's the caller's responsibility to save any values it will need when it resumes.

To do a procedure return, it is only necessary to copy the register values out of the continuation that's pointed to by the `cont` register. This will restore the caller's environment and its pointer to its caller's continuation, and setting the `PC` register will branch to the code in the caller where execution should resume. We often call this "popping" a continuation, because it's a stacklike operation—saving a (partial) continuation pushes the values in registers onto the front of the "stack," and restoring one pops the values back into the registers. (As we will explain later, however, Scheme continuation chains don't always observe this simple stack discipline, which is why they can't be implemented efficiently as contiguous arrays.)

If we save state and do a procedure call, and before returning our caller saves *its* state and does a procedure call, the chain of continuations gets longer. For the most part, this is like pushing activation information on a stack.

```
                                    /|\
                                     |
                    +---------+      |
                    | p.cont. |      |
                    +=========+      |
               cont |    +----+------+
                    +---------+
               envt |    +----+------->old envt
                    +---------+
                 pc |    +----+------->return address
                    +---------+
                              |
                    +   ...
                    |         |
                    +---------+

                               _
                              |\
                               \
                                \
                                 \
                                  \
                                   .
                    +---------+    |
     +-------+      | p.cont. |    |
cont |   +---+----->+=========+   /
     +-------+ cont |    +----+--'
                    +---------+
               envt |    +----+------->old envt
                    +---------+
                 pc |    +----+------->return address
                    +---------+
                              |
                    +   ...
                    |         |
                    +---------+
```

Notice that when we say we save the "state" of the caller, we mean the values in our important registers, but we don't directly save particular variable values—when we save the environment pointer, we don't save the values in the bindings in the environment. If other code then executes in that same environment and changes those values, the new values will be seen by this procedure when it returns and restores the environment pointer. This policy has two important consequences:

1. we can save an environment pointer into a continuation very quickly, and restore it quickly, because we're just saving and restoring one pointer, and
2. it ensures that environments have the right semantics: closures that live in the same environment *should* see each others' changes to variables. This is one of the ways that procedures are supposed to be able to communicate—by operating on variables that they can see.

Executing a return ("popping a continuation") does not modify the partial continuation being popped—it just involves getting values out of the continuation and putting them into registers. Continuations are thus created and used nondestructively, and the continuations on the heap form a graph that reflects the pattern of non-tail procedure calls. Usually, that graph is just a tree, because of the tree-like pattern of call graphs, and the current "stack" of partial continuations is just the rightmost path through that graph, i.e., the path from the newest record all the way back to the root.

Consider the following procedures, where `a` calls `b` twice, and each time `b` is called, it calls `c` twice:

```
(define (a)
   (b)
   (b)
   #t)

(define (b)
   (c)
   (c)
   #t)

(define (c)
   #f)
```

All of these calls are non-tail calls, because none of the procedures ever ends in a (tail) call.

Suppose we call `a` after pushing a continuation for `a`'s caller, then `a` calls `b` the first time. `a` will push a continuation to save its state, then call `b`. While executing `b`, `b`'s state will be in the registers, including a pointer to the continuation for `a` in the `CONT` register.

```
               cont for a's caller
                   /
                 /
          cont. for a
              /|\
      +---+    |
 CONT | +-+----+
      +---+
```

`b` will then push a continuation and call `c`.

```
               cont for a's caller
                   /
                 /
          cont. for a
              /
            /
     cont. for b
        /|\
         |
      +-|-+
 CONT | + |
      +---+
```

When `c` returns, it will restore `b`'s state by popping the partial continuation's values into registers. At this point, the `CONT` register will point past the continuation for `b` to the continuation for `a`.

```
               cont for a's caller
                    /
                   /
           cont. for a
              /  /|\
             /     |
    cont. for b    |
                   |
      +---+        |
 CONT | +-+--------+
      +---+
```

Then b will push another continuation and call c again.

```
               cont for a's caller
                    /
                   /
          cont. for a
              /  \
             /     \
    cont. for b  cont for b
                    /|\
                     |
      +---+          |
 CONT | +-+----------+
      +---+
```

Again, c will return, restoring b's state, and the CONT register will point past the continuation for b to the continuation for a.

```
              cont for a's caller
                   /
                 /
         cont. for a <-------+
             /  \            |
           /     \           |
   cont. for b  cont for b   |
                             |
     +---+                   |
CONT | +-+-------------------+
     +---+
```

After returning to a, the CONT register will point past the continuation for a to the continuation for a's caller. Then before a calls b again, it will push another continuation to save its state.

```
              cont for a's caller
                   /        \
                 /            \
         cont. for a        cont for a
             /  \              /|\
           /     \              |
   cont. for b  cont for b      |
                                |
     +---+                      |
CONT | +-+----------------------+
     +---+
```

Then a will return and the CONT register will point past the continuation for a to the continuation for a's caller.

```
                cont for a's caller <--+
                    /                  |
                   /                   |
             cont. for a               |
                 /  \                  |
               /     \                 |
       cont. for b  cont for a         |
                                       |
         +---+                         |
    CONT | +-+-------------------------+
         +---+
```

This continues in the obvious way, so that at the time of the fourth and last call to C, the continuations on the heap look like this:

```
                cont for a's caller
                    /          \
                   /             \
             cont. for a         cont. for a
                 /  \                /  \
               /     \             /      \
       cont for b  cont for b  cont for b  cont for b
                                             /|\
                                              |
         +---+                                |
    CONT | +-+--------------------------------+
         +---+
```

Most of the time, the rest of this graph becomes garbage quickly—each continuation holds pointers back up the tree, but nothing holds pointers *down* the tree. Partial continuations therefore usually become garbage the first time they're returned through.

The fact that this graph is created on the heap will allow us to implement `call-with-current-continuation`, a.k.a. `call/cc`, a very powerful control construct. `call/cc` can capture the control state of a program at a particular point in its execution by pushing a partial continuation and saving a pointer to it. Later, it can magically return control to that point by restoring that continuation, instead of the one in the continuation register. (We will discuss `call/cc` in detail in Chapter XX.)

## 6.3 Exploiting Tail Recursion

In an earlier section, we presented example recursive implementations of several Scheme functions; some of them were tail recursive, and some not.

At first glance, many routines seem as though they can't conveniently be coded tail recursively. On closer inspection, many of them can in fact be coded this way.

### 6.3.1 Passing Intermediate Values as Arguments

#### 6.3.1.1 Summing a List

Suppose we want to sum a list of numbers. The most obvious way of doing it is the way we did it earlier, like this:

```
(define (list-sum lis)
   (if (null? lis)
       0
       (+ (car lis)
          (list-sum (cdr lis)))))
```

The problem with this code is that it's not particularly efficient, because it's not tail recursive. After each recursive call to `list-sum`, we must return to do the addition that adds one element to the sum of the rest of the list. We're adding the elements of the list back-to-front, on the way back up from nested recursion. (This means that Scheme must push a partial continuation before every recursive call, and each one must be popped when we're finished, to return the sum back from each call to its caller.)

We can write a tail-recursive version of `list-sum` that adds things in front-to-back order instead. The trick is to do the addition before the tail call, and to *pass the sum so far to the recursive call*, i.e., to pass it *forward* as an argument until a non-tail call returns it.

To do this, we have to keep a running sum, and each recursive call must pass it as an argument to the next. To start it off, we have to have a "running sum" of 0.

We can do this by defining two procedures. The one that does the real work takes a list and a running sum, adds one element to the running sum, and tail-calls itself to add the rest of the elements to the running sum. When it reaches the end of the list, it just returns the value. (Scheme doesn't need to save a partial continuation before each call, since only the last call ever returns.)

For convenience, we also wrap this procedure up in a friendlier procedure that will start off the recursion, by supplying an initial "running sum" of 0.

```
; a tail-recursive list summing procedure
(define (lsum lis sum-so-far)
   (cond ((null? lis)
          sum-so-far)
         (else
          (lsum (cdr lis)
                (+ sum-so-far (car lis))))))

; a friendly wrapper that supplies an initial running sum of 0
(define (list-sum lis)
   (l-s-aux lis 0))
```

We can make this cleaner by encapsulating `lsum`, since it's only used by list-sum. We make `lsum` a local procedure using `letrec` and `lambda`.

```
(define (list-sum lis)
   ; define a local, tail-recursive list summing procedure
   (letrec ((lsum (lambda (lis sum-so-far)
                    (cond ((null? lis)
                           sum-so-far)
                          (else
                           (lsum (cdr lis)
                                 (+ sum-so-far (car lis))))))))
      (lsum lis 0))) ;; start off recursive summing with a sum of 0
```

We can write this more clearly using named `let`:

```
(define (list-sum lis)
   (let loop ((lis lis)
              (sum-so-far 0))
      (cond ((null? lis)
             sum-so-far)
            (else
             (loop (cdr lis)
                   (+ sum-so-far (car lis)))))))
```

Notice that here we're using two loop variables, rebound at each iteration. One keeps track of the remaining part of the original list, and the other the sum of the list items we've seen so far.

Also notice that the version using named `let` is exactly equivalent to the version using explicit tail-recursion.

#### 6.3.1.2 Implementing `length` tail-recursively

Recall that in [ Chapter whatever ] we implemented `length` this way:

```
(define (length lis)
   (if (null? lis)
       0
       (+ 1 (length (cdr lis)))))
```

This definition looks a lot like the definition of `list-sum`, and has the same basic problem. By using straightforward recursion (adding one to the length of the rest of the list), we're ensuring the addition happens back-to-front. We can compute the list length front to back by passing the running sum *forward* through tail recursions, as an argument. Each tail call will add to the running sum, and pass it forward. When the last tail call returns to its caller, it just returns the sum.

To do this, it's convenient to write the `length` procedure as a wrapper around a two-argument procedure that passes the running sum (as well as the remainder of list) to recursive calls to itself.

```
(define (length lis)
   (letrec ((len (lambda (lis length-so-far)
                   (if (null? lis)
                       length-so-far
                       (len (cdr lis)
                            (+ 1 length-so-far))))))
     (len lis 0))
```

Or equivalently, using named `let`:

```
(define (length lis)
   (let loop ((lis lis)
              (length-so-far 0))
      (if (null? lis)
          len-so-far
          (loop (cdr lis)
                (+ (car lis) length-so-far)))))
```

### 6.3.2 `reduce`

In this section, I'll give an extended example of the use of higher-order functions to express patterns common to many functions, and customizing general procedures with procedural arguments and closure creation.

Consider the following function to sum the elements of a list

```
(define (list-sum lis)
   (if (null? lis)
       0
       (+ (car lis)
          (list-sum (cdr lis)))))
```

Given this definition,

```
(list-sum '(10 15 20 25))
```

is equivalent to

```
(+ 10 (+ 15 (+ 20 (+ 25 0)))).
```

[ the following couple of examples are now redundant with earlier material... trim and refer back. ]

Now consider a very similar function to multiply the elements of a list, where we've adopted the convention that the product of a null list is 1. (1 is probably the right value to use, because if you multiply something by 1 you get back the same thing—just as if you add something to 0 you get back the same thing.)

```
(define (list-prod lis)
   (if (null? lis)
       1
       (+ (car lis)
          (list-prod (cdr lis)))))
```

Given this definition,

```
(list-prod '(2 3 4 5))
```

is equivalent to

```
(* 2 (* 3 (* 4 (* 5 1))))
```

Given these definitions, you can probably imagine a very similar function to subtract the elements of a list, or to divide the elements of a list. For subtraction, the base value for an empty list should probably be zero, because subtracting zero doesn't change anything. For division it should probably be one.

At any rate, what we want is a single function that captures the pattern

(*op thing1*(*op thing2* ...(*op thingn base-thing*)...))

We can write a higher-order procedure `reduce` that implements this pattern in a general way, taking three arguments: any procedure you want successively applied to the elements of a list, an appropriate base value to use on reaching the end of the list, and the list to do it to.

```
(define (reduce fn base-value lis)
   (if (null? lis)
       base-value
       (fn (car lis)
           (reduce fn base-value (cdr lis)))))
```

This is a very general procedure, that can be used for lots of things besides numerical operations on lists of numbers: it can be used for any computation over successive items in a list.

[ need to check the following couple of examples—they're off the top of my head ]

What does `(reduce cons '() '(a b c d))` do? It's equivalent to `(cons 'a (cons 'b (cons 'c (cons 'd '()))))`. That is, `(reduce cons '()` *list*`)` copies a list. We could define `list-copy` that way:

```
(define (list-copy lis)
   (reduce cons '() lis))
```

We could also define `append` that way, because `reduce` allows you to specify what goes at the end of a list—we don't have to end our list with `'()`. Here's a two-argument version of `append`:

```
(define (append list1 list2)
   (reduce cons list2 list1))
```

The reduction of a list using (lambda (x rest) (cons (* x 2) rest)) constructs a new list whose elements are twice the values of the corresponding elements in the original list.

```
Scheme>(reduce (lambda (x rest)
                  (cons (* x 2) rest))
               '()
               '(1 2 3 4))
(2 4 6 8)
```

The `reduce` procedure above is handy, because you can use it for many different kinds of computations over different kinds of lists values, as long as you can process the elements (and construct the result) front-to-back. It's a little awkward, though, in that each time you use it, you have to remember the appropriate base value for the operation you're applying to a list.

Sometimes it would be preferable to come up with a single specialized procedure like `list-sum`, which implicitly remembers which function it should apply to the list elements (e.g., `+`) and what base value to return for an empty list (e.g., 0).

We can write a procedure `make-reducer` that will automatically construct a reducer procedure, given a function and a base value. Here's an example usage:

```
Scheme> (define list-sum (make-reducer + 0))
list-sum

Scheme> (define list-product (make-reducer * 1))
list-copy

Scheme> (list-sum '(1 2 3 4))
10

Scheme> (list-product '(1 2 3 4))
24
```

Make sure you understand the expressions above. The define forms are `not` using procedure definition syntax—they're using plain variable definition syntax, but the initial value expressions return procedures constructed by make-reducer. If we hadn't wanted to define procedures named list-sum and list-product, and hang on to them, we could have just taken the procedures returned by make-reducer and called them immediately:

```
Scheme> ((make-reducer + 0) '(1 2 3 4))
10

Scheme> ((make-reducer * 1) '(1 2 3 4))
24
```

This is very much like calling our original `reduce` procedure, except that each time we're constructing a specialized procedure that's like `reduce` customized for particular values of its first two arguments; then we call that new, specialized procedure to do the work on a particular list.

Here's a simple definition of `make-reducer` in terms of `reduce`:

```
(define (make-reducer fn base-value)
   (lambda (lis)
      (reduce fn base-value lis)))
```

Notice that we *are* using procedure definition syntax here, so the `lambda` in the body will create and return a closure.

But suppose we don't already have a reduce procedure, and we don't want to leave one lying around. A cleaner solution is to define the general `reduce` procedure as a local procedure, and create closures of it in different environments to customize it for different functions and base values.

```
(define (make-reducer fn base-value)
   (letrec ((reduce (lambda (lis)
                       (if (null? lis)
                           base-value
                           (fn (car lis)
                               (reduce (cdr lis)))))))
        reduce)) ; return new closure of local procedure
```

This procedure uses closure creation to create a customized version of `reduce` When `make-reducer` is entered, its arguments are bound and initialized to the argument values—i.e., the function and base value we want the custom reducer to use. In this environment, we create a closure of the standard reducer procedure using lambda. We wrap the lambda in a letrec so that the reducer can see its own name. Notice that since reduce is a local procedure, it can see the arguments to `make-reducer`, and we don't have to pass it those arguments explicitly.

By using local procedure definition syntax—which not all Schemes support—we can write this as:

```
(define (make-reducer fn base-value)
   (define (reduce lis)
      (if (null? lis)
          base-value
          (fn (car lis)
              (reduce (cdr lis)))))
   reduce))  ;return new closure of local procedure
```

Make sure that you understand that these are equivalent—the local procedure `define` is equivalent to a `letrec` and a `lambda`, and in either case the closure created (by the `lambda` or the `define`) will capture the environment where the arguments to `make-reducer` are bound.

# 7 Quasiquotation and Macros

Scheme provides facilities for transforming expressions automatically to create new expressions. These facilities are called *quasiquotation* and *syntax extension* (or "macros"). Transformational programming is one of the most powerful features of Scheme.

Quasiquotation allows you to specify patterns that can be used to construct data structures, and also specify how to fill in "holes" in the patterns. In effect, you can define a template for a data structure, much like a quoted data structure, but also specify how to fill in holes to create variations on the data structure.

Syntax extension allows you to do something very similar for code. You can write "macros" that specify most of an expression, and you can fill in the holes in these templates to create particular expressions. With macros, you can write "templates" for programs, which you can customize by filling in the holes. This lets you create both code-structuring and data-structuring facilities that express stereotyped patterns with variations.

[ Scheme macros are actually more powerful than this, however, because you can use them to analyze code before transforming it... sort of... ]

With Scheme macros, you can define new control constructs, data structuring facilities, full-blown object systems with inheritance, parameterized coding facilities (like C++ templates), and other more application-specific facilities to make your life easier.

## 7.1 `quasiquote`

The special form `quasiquote` behaves a lot like `quote`, allowing you to write out literal expressions in your program, using the standard textual representation of s-expressions. Scheme automatically constructs the data structures. `quasiquote` is much more powerful than `quote`, however, because you can write expressions that are *mostly* literal, but leave holes to be filled in with values computed at runtime.

For example, the value of the expression `(quote (foo bar baz))` is a list `(foo bar baz)`. Likewise, the value of the expression `(quasiquote (foo bar baz))` is a list `(foo bar baz)`.

There's a big difference, though. `quote` constructs an s-expression at compile time, when the procedure containing the `quote` expression is compiled.[1] `quasiquote` constructs an s-expression at *run time*, when the `quasiquote` form is executed. This allows Scheme to "customize" a data structure, so that you actually get a *different* data structure each time you execute the same `quasiquote` form. You can use the `unquote` operator to specify which parts should be customized.

For example, suppose you want to write a procedure that creates a three-element list whose first and last elements are the literal symbols `foo` and `baz`, but whose middle element is the `value` of the variable `bar`.

Try this in your scheme system:

```
Scheme>(define bar 2)
baz
Scheme>(quasiquote (foo (unquote bar) baz))
(foo 2 baz)
```

Without `quasiquote` and `unquote`, you could get the same effect by replacing `(quasiquote (foo (unquote bar) baz))` with `(list (quote foo) bar (quote baz))`, or the equivalent sugared form `(list 'quote foo 'baz)`. For this simple example, that's probably at least as clear, because the use of `(quasiquote ...)` and `(unquote ...)` is rather clunky.

To make it easier to write quasiquoted expressions, Scheme provides a little syntactic sugar. Just as you can use a single quote character and write `'(foo bar baz)` instead of `(quote (foo bar baz))`, you can use a *backquote* character (`` ` ``) to replace `(quote ...)` and a *comma* character (`,`) to replace `(unquote ...)`.

Now we can do this:

```
Scheme>`(foo ,bar baz)
(foo 2 baz)
```

[1] This is something of an oversimplification. The quoted expression is generally converted to a list at `read` time, and the value of the `quote` expression may be the very same data structure, or it may be a copy made later during compilation or interpretation. At any rate, you should only count on the *structure* of a quoted data structure, not its identity. A compiler or interpreter is allowed to return the same list for different `quote`d data structures with the same structure.

This is much clearer. Intuitively, the backquote character means "construct an s-expresson of the following (literal) form, except where commas appear," and the comma character means "use the *value* of the following expression here, instead of using it literally."

Now you can see why it's called `quasiquote`—it's a way of writing "mostly quoted" expressions, instead of pure literals. You can turn quoting off where you want to. This is particularly useful in constructing s-expressions that are in fact *mostly* literal, especially if they're complicated.

For a simple example, suppose you want to write a procedure that constructs a greeting to print to a user. The greeting is always mostly the same, but includes the current day of the week:

```
Scheme> (define day-of-week 'Sunday)
day-of-week

Scheme> (define (make-greeting)
              `(Welcome to the FooBar system!  We hope you
                enjoy your visit on this fine ,day-of-week)))
greet


Scheme>(make-greeting)
(Welcome to the FooBar system! We hope you enjoy your visit on this
fine Sunday)

Scheme>(set! day-of-week 'Monday)
day-of-week

Scheme>(make-greeting)
(Welcome to the FooBar system! We hope you enjoy your visit on this
fine Monday)
```

You may have notice that this is somewhat similar to formatted output in other languages you've used, like C. (C's `printf` procedure takes a string that is (mostly) quoted, but has special escape characters in it to tell where to substitute the printed representation of runtime values. For example, if `day_of_week` holds a pointer to the string `"Sunday"`, `printf("Welcome. It's %s.", day_of_week)` prints `"Welcome. It's Sunday."`)

The nice thing about Scheme quasiquotation is that it works on normal data structures. For example, suppose you want to write a routine that creates an association list with several literal elements, and a several customized ones.

```
(define (create-shipping-employee-association name)
  `((name ,name)
    (employee-id-no ,(get-next-employee-id!))
    (department shipping)
    (hire-date ,(get-day) ,(get-month) ,(get-year))))
```

(Notice that here that most of the unquoted expressions are calls to procedures, whose return values will be used. We can fill the holes in our templates with anything we want, not just variable values.)

Depending on the value of the variable the values returned by the procedure calls, `(new-shipping-employee-alist "Philboyd Studge")` will return something like

```
((name "Philboyd Studge")
 (employee-id-no 6357)
 (department shipping)
 (hire-date 18 August 1997))
```

Here it should be clear that `quasiquote` has let us write out a stereotyped data structure, and `unquote` lets us fill in the varying parts. More complicated examples would be make this benefit clearer, but I'll leave them to your imagination.

### 7.1.1 unquote-splicing

Scheme provides a variant of `unquote` for use when you want to merge an unquoted list into a literal list, rather than nesting it.

For example, suppose you want to embed a phrase in a sentence, where the phrase is a list of symbols, and the sentence is a list of symbols.

If you tried this with unquote, you'd get a nested list, rather than just a list of symbols:

```
Scheme> (define phrase-of-the-day '(the Lord helps those who take a big
                                helping for themselves))
phrase-of-the-day

Scheme> `(Remember that ,phrase-of-the-day)
(Remember that (the Lord helps those who take a big helping for
themselves))
```

Rather than using `,`*expr*), we can use use (`unquote-splicing` *expr*), or the syntactically sugared form, `,@`*expr*.

```
Scheme> `(And remember that ,@phrase of the day)
(And remember that the Lord helps those who take a big helping for
themselves)
```

## 7.2 Defining New Special Forms

Sometimes we want to write stereotyped *code*, not just stereotyped data structures. As with data, we sometimes want part of our stereotyped piece of code to vary. We can do this with *syntax extensions*, also known as macros.

(If you're familiar with macros from C, don't scoff. Macros in C are stunningly lame and hard to use compared to Lisp or Scheme macros. Read on to find out what you've been missing. If you're familiar with Lisp macros, but have never done advanced programming with them, you probably don't realize how powerful they are—Lisp macros are so error-prone that people often avoid them. Scheme macros are very powerful, but automate away some of the tricky parts.)

Macros are syntax extensions to a programming language, expressed as a translation of expressions. By writing a macro, what you're really doing is extending the functionality of the compiler or interpreter—you're telling it how to compile (or interpret) a new construct, by telling it how to rewrite that construct into something it already knows how to compile or interpret.

(Conceptually, defining a macro is extending the compiler—you're telling the parser how to recognize a new construct, to change the grammar of the language, and also specifying how to generate code for the new construct. This is something you can't do in most languages, but it's easy in Scheme.)

Scheme recognizes macro definitions, and then uses them to recognize and translate the new constructs into other constructs. The interpreter or compiler's process of translating a level constructs is often called "macro expansion," despite the fact that the resulting expression may not be bigger than the original expression. Macroexpansion can be recursive, because macros can use macros, and a macro can even use itself, like a recursive procedure.

Syntax extension is powerful, and hence somewhat dangerous when used too casually. Be aware that when you write a macro, you can change the syntax of your programming language, and that *can* be a bad thing—you and others may no longer be able to easily understand what the program does. Used judiciously, however, such syntactic extensions are often just what you need to simplify your programs. They are especially useful for writing programs that write programs, so that you can avoid a lot of tedious repetitive coding.

Macros are so useful that they're usually used in the implementation of Scheme itself. Most Scheme compilers actually understand only a few special forms, and the rest are written as macros.

In a later chapter, I'll describe some advanced uses of macros, which let your "roll your own" language with powerful new features.

### 7.2.1 Macros vs. Procedures

Why do we want macros? In Scheme, the main code abstraction mechanism is procedural abstraction, e.g. using `define` or `lambda` to write procedures that do stereotyped things. In a sense, we "specialize" procedures by passing them argument values—a procedure can do different things depending on the values it's given to work with. We can also "specialize" procedures by creating closures in different environments. Isn't this enough?

Not in general. While procedural abstraction is very powerful, there are times when we may want to write stereotyped routines that can't be written as procedures.

Suppose, for example, you have a Scheme system which gives you things like `let` and `if`, but not `or`. (Real Schemes all provide `or`, but pretend they don't. It makes a nice, simple example.)

You want an `or` construct (rather like the one actually built into Scheme). This `or` can take two arguments; it evaluates the first one and returns the result if it's a true value, otherwise it evaluates the second one and returns that result.

Notice that you can't write `or` as a procedure. If `or` were a procedure, both of its arguments would always be evaluated *before* the actual procedure call. Since `or` is only supposed to evaluate its second argument if the first one returns `#f`, it just wouldn't work.

If Scheme didn't have `or`, you could fake it at any given point in your program, by writing an equivalent `let` expression with an `if` statement in it.

For example, suppose you wanted to write the equivalent of `(or (foo?) (bar?))`.

As a first try, you might do this:

```
(if (foo?)
    (foo?)
    (bar?))
```

That is, test `(foo?)`, and return its value if it's a true value. That's not really quite right though, because this if statement evaluates `foo?` twice: once to test it, and once to return it.

We really only want to evaluate it once—if `(foo?)` is an expression with side effects, evaluating it twice could make the program incorrect as well as inefficient.

To avoid this, we can always evaluate the first expression just once to get the value, and store that value in a temporary variable so that we can return it without evaluating it again:

You could instead write

```
(let ((temp (foo?)))
   (if temp
       temp
       (bar?)))
```

This `let` expression gives the same effect as `(or (foo?) (bar?))`, because it evaluates `foo` exactly once, and then tests the value; if the value is true, it returns that value. (The use of a `let` variable to stash the value allows us to test it without evaluating `(foo?)` again.) If the value is `#f`, it evaluates `(bar?)` and returns the result.

This is the transformation we'd like to be able to automate by defining `or` as a macro.

Here's a simple version of `or` written as a macro. I've called it `or2` to distinguish it from Scheme's normal `or`.

```
(define-syntax or2
   (syntax-rules ()
      ((or2 a b)        ; pattern
       (let ((temp a)) ; template
          (if temp
              temp
              b)))))
```

What we're saying to Scheme is that we want to define the syntax of `or` by giving *syntax rules* for recognizing it and translating it. For this simple version of `or`, we only need one rule, which says to translate `(or` *a b*`)` into the desired `let` expression.

`(or a b)` is called the rule's *pattern*, which specifies what counts as a call to the `or` macro, and the `let` expression is the rule's *template*, which specifies the equivalent code that Scheme should translate calls into.

The variables `a` and `b` are called *pattern variables*. They stand for the actual expressions passed as arguments to the macro. They are "matched" to the actual expressions when the pattern is recognized, and when the template is interpreted or compiled, the actual expressions are used where the pattern variables occur.

You can think of this in two steps. When a macro is used,

1. the template is copied, except that the pattern variables are replaced with the macro's argument expressions,
2. the result is interpreted (or compiled) in place of the call expression.

(It's really not quite this simple, but that's the basic idea.)

In some ways, macro arguments are a lot like procedure arguments, but in other ways they're very different. The pattern variables are *not* bound at run time, and don't refer to storage locations. They're only used in translating a macro call into the equivalent expression.

Always remember that arguments to a macro are *expressions* used in transforming the code, and *then* the code is executed. (For example, the output of the `or` macro doesn't contain a variable named `a`; `a` is just a shorthand for whatever expression is passed as an argument to the macro. In the example use `(or (foo?) (bar?))`, the expression `(foo?)` is what gets evaluated at the point where `a` is used in the macro body.)

This is why our macro has to use a temporary variable, like the hand-written equivalent of `or`. If we tried to write the macro like a procedure, without using a temporary variable, like this

```
(define-syntax or
   (syntax-rules ()
      ((or a b)
       (if a
           a
           b))))
```

then `(or (foo?) (bar?))` would be translated into

```
(if (foo?)
    (foo?)
    (bar?))
```

As with the buggy handwritten version, `(foo?)` would be evaluated twice when this expression was evaluated.

(This is the most common mistake in writing macros—forgetting that while macros give you the ability to control when argument expressions are evaluated, they also *require* you to control it. It's safe to use a procedure argument multiple times, because that's just referring to a value in a run-time binding. Using a macro argument causes evaluation of the entire argument expression at that point.)

We can make a better `or` by using more rules. We might want `or` to work with any number of arguments, so that

1. `or` of zero arguments returns `#f`, because it has zero true arguments,
2. `or` of one argument is equivalent to that argument—it's true if and only if that argument is true.
3. `or` of two or more arguments evaluates its first argument, and returns its value if it's true. Otherwise, it computes the `or` of the rest of its arguments, and returns its result.

Here's the Scheme definition with these three rules:

```
(define-syntax or
   (syntax-rules ()
      ((or)             ; OR of zero arguments
       #f)              ;   is always false
      ((or a)           ; OR of one argument
       a)               ;   is equivalent to the argument expression
      ((or a b c ...)   ; OR of two or more arguments
       (let ((temp a))  ;   is the first or the OR of the rest
          (if temp
              temp
              (or b c ...))))))
```

Notice that this definition is recursive. (The third rule's template uses the `or` macro recursively.) If we hand `or` four arguments, like this: `(or foo bar baz quux)`, it should be equivalent to `(or foo (or bar (or baz (or quux))))`.

Scheme will use recursion to translate the expression, one step at a time. When Scheme encounters a macro call, it transforms the call into the equivalent code, using the appropriate rule. It then interprets (or compiles) the resulting expression. If the result itself includes a macro call, then the interpreter (or compiler) calls itself recursively to translate *that* before evaluating it. For a correctly written macro, the recursive translation will eventually "bottom out" when no more macro calls result, and the code will be evaluated in the usual way.

(As I'll show later, this fits in very neatly with the interpreter's or compiler's recursive evaluation of expressions.)

This recursion is recursion in Scheme's *transformation* of the call expression into equivalent code—it doesn't mean that the resulting code is recursive. A Scheme compiler will do all of the recursive transformation at compile time, so there's no runtime overhead. Of course, the recursion has to terminate, or the compiler will not be able to finish the translation.

In this definition of `or`, the third rule contains the symbols `c ...`. The Scheme identifier `...` is treated specially, to help you write recursive rules. (In previous examples, I used ... as an ellipsis to stand for code I didn't want to write out, but here we're usuing the *actual Scheme identifier* `...`; it's actually used in the Scheme code for macros.)

Scheme treats a pattern variable followed by `...` as matching `zero or more` subexpressions. In this `or` macro, `c ...` matches *all* of the arguments after the first two.

Scheme matches `(or foo bar baz quux)` by the third rule, whose pattern `(or a b c ...)`, because it has at least two arguments. In applying the rule, Scheme matches `a` to `foo`, `b` to `bar`, and `c ...` to the *sequence* of expressions `baz bleen`.

[ This is similar to how you use `unquote-splicing` inside backquote—you can splice a list into a list at the same level, rather than nesting it. ]

The result of processing this `or` is

```
(let ((temp foo))
      (if temp
          temp
          (or bar baz quux)))
```

Now Scheme evaluates this expression.

But there's another `or` in there—when Scheme gets to `(or bar baz quux)` it will match the third rule again, with `a` matched to `bar`, `b` matched to `baz`, and `c ...` being matched to just `quux`. The result of this macro-processing step is

```
(let ((temp foo))
   (if temp
       temp
       (let ((temp bar))
          (if temp
              temp
              (or baz quux)))))
```

And the new let expression is evaluated.

There `or` is again, so Scheme will treat `(or baz quux)` the same way, again using the third rule—this time matching `a` to `baz`, `b` to `quux`, and `c ...` to nothing at all, producing

```
(let ((temp foo))
   (if temp
       temp
       (let ((temp bar))
          (if temp
              temp
              (let ((temp baz)
                 (if temp
                     temp
                     (or quux))))))))
```

And this will be evaluated.

Now the resulting `or` matches the *second* rule in the `or` macro, because it has only one argument `quux`, which is matched to `a`. The whole translation is therefore:

```
(let ((temp foo))
   (if temp
       temp
       (let ((temp bar))
          (if temp
              temp
              (let ((temp baz)
                 (if temp
                     temp
                     quux)))))))
```

There are no more macro calls here, so the recursion terminates.

You might have noticed that the example translation of `(or foo bar baz quux)` has several different variables named `temp` in it. You might have wondered if this could cause problems—is there a potential for accidentally referring to the wrong variable in the wrong place in the code generated by a macro?

The answer no. Scheme's macro system actually does some magic to avoid this, which I'll discuss later in a later section. Scheme actually keeps track of which variables are introduced by different applications of macros, and keeps them distinct—the different variables named `temp` are treated as though they had different names, so that macros follow the same scope rules as procedures. (Scheme macros are said to be *hygienic*; what that really means is that they respect lexical scope.)

You can think of this as a *renaming*, as though Scheme had sneakily changed the names each time the macro was applied to transform the expression, and the result were

```
(let ((temp_1 foo))
   (if temp_1
       temp_1
       (let ((temp_2 bar))
          (if temp_2
              temp_2
              (let ((temp_3 baz)
                 (if temp_3
                     temp_3
                     quux)))))))
```

Scheme implements the same scoping rule for macros and their arguments as for procedures and their arguments. When you call a procedure, the argument expressions are evaluated at the call site, i.e., in the call site environment, and the values are passed to the procedure—the environment inside the called procedure doesn't affect the meaning of the argument expressions. Likewise

In writing macros like `or`, we want to control *when* and *whether* the arguments are evaluated, but otherwise we want them to mean the same thing they would if they were arguments to a procedure.

For example, suppose we call `or` with an argument expression that happens to use a name that's used inside `or`. `or` uses a local variable named `temp`, and we might just happen to pass it an expression using the name `temp`.

Consider the following procedure, which uses local variables `perm` and `temp`, and calls `or` in their scope.

```
(define (employee? person)
  (let ((perm (member person permanent-employees))
        (temp (member person temporary-employees)))
     (or perm temp))
```

If we translated the or macro naively, without worrying about accidental naming conflicts, we'd get this:

```
(define (employee? person)
  (let ((perm (member person permanent-employees))
        (temp (member person temporary-employees)))
     (let ((temp perm)
        (if temp
            temp
            temp)))
```

(This is *not* what R5RS Scheme macros do!)

Note what's wrong here. The name `temp` was passed into the macro from the call site, but it appeared in the body of the macro inside the `let` binding of `temp`. At the call site, it referred to the "outer" `temp`, but inside the macro, it turne out to refer to something else—in the process of moving the expression around, we accidentally changed its meaning.

## 7.3 Implementing More Scheme Special Forms

As examples of Scheme macros, I'll show how to implement several special forms in terms of `lambda`. This is how most real Scheme compilers work—the compiler itself only "understands" how to compile a few special forms directly, but the others can be defined as macros.

Traditionally, the compiler understands `lambda`, and all other binding forms are implemented in terms of `lambda` and procedure calling. The compiler must also understand a few other special forms, `if`, `set!`, `quote`, a simple version of `define` [ did I leave one out? ].)

### 7.3.1 `let`

Recall that in chapter [ whatever ], I explained how the semantics of `let` can be explained in terms of `lambda`. For any `let` expression, which binds variables and evaluates body expressions in that scope, there is an exactly equivalent expression using `lambda` and procedure calling. The `lambda` creates a procedure which will bind the variables as its argument variables, and execute the body of the `let`. This `lambda` is then used in a combination—calling the procedure makes it bind variables when it accepts arguments.

```
(define-syntax let ()
   (syntax-rules
     ((_ ((var value-expr) ...) body-expr ...)  ; pattern
      ((lambda (var ...)
          body-expr ...)
       (value-expr ...)))))
```

Here I've used an underscore to stand for the keyword `let` in the macro call pattern. This is allowable, and recommended, because it avoids having to write the keyword in several places. (If you had to write out the keyword in each pattern, it would make it more difficult and error-prone to change the name of a macro.)

I've also taken advantage of the fact that Scheme is pretty smart about patterns using the `...` (ellipsis) symbol. The pattern has two ellipses. One matches any number of binding forms (variable names and initial value expressions); the other matches any number of body expressions.

The body expressions matched by `body-expr ...` are simply used in the body of the `lambda` expression.

The expressions matched by `(var value-expr) ...` are used differently, however—they are not simply substituted into the macro template. Instead, `(var ...)` is used to generate the argument list for the `lambda`, and `value-expr ...` is used to generate the list of initial expressions.

Scheme's macro system is smart enough to figure out what's going on. If the pattern contains an ellipsis following a compound form (like `(var init-expr) ...`, and the template uses one of the pattern variables from that compound form (followed by an ellipsis), then Scheme assumes you want *the corresponding part* of each form matched by the pattern form.

If we think of the expressions as s-expressions, we've matched a pattern that is one list of two-element lists, and restructured it into two separate lists of elements. (That is, we're going from a list of `car`s and `cadr`s to a list of `car`s and a list of `cadr`s.)

As an example of use, consider

```
(let ((var-a (some-procedure foo))
      (var-b (some-procedure bar)))
   (quux var-a)
   (quux var-b))
```

which translates to

```
((lambda (var-a var-b)
    (quux var-a)
    (quux var-b))
 (some-procedure foo)
 (some-procedure bar))
```

[ The following is out of place—here I should just be showing some uses of macros. The problem is that I don't want to lie and pretend that it's all very simple—Scheme does something sophisticated when you write binding contstructs as macros...

This stuff will all be clearer after I've talked about hygiene problems with Lisp macros, and laziness and call-by-name... how to fwd ref gracefully? ]

An extraordinarily astute and thoughtful reader might wonder if there's something wrong here. (Luckily, there's actually nothing to worry about.) Recall that when discussing `or`, I said that Scheme is careful to treat names introduced by a macro as though they were distinct, effectively renaming variables introduced in a macro. What about the argument variables to `lambda` in this example? One might think `var-a` and `var-b` would just be renamed and we'd get:

```
((lambda (var-a-1 var-b-1)
    (quux var-a)
    (quux var-b))
 (some-procedure foo)
 (some-procedure bar))
```

Clearly, this isn't what we want—we *want* `var-a` and `var-b` in the `lambda` body to refer to the variables introduced in by `lambda`—that's what it's for.

Scheme's macro processor is smart enough to infer that this is what you want. When you write a macro that accepts a *name* as an argument and *binds* it, Scheme assumes you're doing that for a good reason. If you then take another argument to the same macro and use it in the scope of that new variable, Scheme assumes you want occurrences of the name to refer to the `new` variable.

That is, Scheme uses an algorithm that checks what you do with names in forms that get passed as arguments into a macro. If you just use them in the normal ways, evaluating or assigning to them as variable names, Scheme assumes you mean to refer to whatever those names refer to at the call site of the macro. (That's normal lexical scope.) But if you take the name and use it `as` the name of a new variable, Scheme assumes you're defining a binding construct, and that any other arguments you put in that scope should see the new binding, instead of being scoped at the call site.)

Scheme can generally assume this, because if you're not implementing a scoping binding form (like let or do), there's no reason for a macro to accept a name as an argument and then turn around and bind it.

### 7.3.2 let*

Once we have let, we can implement let* in terms of that. We simply write a recursive macro that peels off one binding form at a time and generates a let, so that we get a nested set of lets that each bind one variable.

```
(define-syntax let* ()
   (syntax_rules
      ((_ () body-expr ...)
       (begin body-expr ...))
      ((_ ((var1 value-expr1)(var value-expr) ...)
       (let ((var1 value-expr))
          (_ ((var value-expr) ...)
             body-expr ...)))))
```

This macro uses two syntax rules. The first is the termination case for recursive macroexpansion. A let* that has an empty binding form (i.e., binds zero variables) should be translated into a begin; it will just sequence the body expressions.

The recursive rule says that a let* with one or more binding subforms should translate into a let that performs the first binding and another let* to bind the rest and evaluate the body. (Note that I've used the _ shorthand for let* in the recursive call, as well as in the pattern.)

As with let, Scheme recognizes this as a binding construct, and does the right thing—it notices that the var argument passed into the macro is used as a name of a new binding in the macro, so it assumes that the new binding should be visible to the body expressions.

### 7.3.3 cond

### 7.3.4 Discussion

Scheme macros also have several features I haven't demonstrated, to make it easier to write more sophisticated macros than `or`, and I'll demonstrate those later, too.

In the next section, though, I will discuss a different and simpler kind of macro system, which is *not* standard Scheme, and *does* have problems with variable names.

## 7.4 Lisp-style Macros

In this section, I'll discuss a simple kind of macro system that isn't standard in Scheme (and you might be able to skim this section without too much loss) but is interesting for several reasons.

- It is very easy to explain how it works—it is a real macro system, but one which is very easy to implement. We can add it to our interpreter with a few function definitions. This should clear up any confusion about what macros basically are, and how to think about them. (It's also another nice example of Scheme programming—we'll get to cheat and use `quasiquote` to do most of our work for us. Then I'll show how to implement `quasiquote`, too.)
- The simple Lisp-style macro system also demonstrates two important issues in macros: the power of procedural transformation, and problems with scoping when code is transformed. An understanding of Lisp macros can only help later when we return to Scheme macros for an in-depth discussion of how to work and how to use them.
- The new standard Scheme macro system is safer than Lisp macros, and very useful, but not quite as powerful. Sometimes it's they're still useful, if you use them for simple things they're appropriate for. Some of our later examples will use this kind of macro.
- 

  [ R5RS will have macros, but IEEE/ANSI Scheme does not, and may not for some time. Most Schemes do support Lisp-style macros, even though they're not part of the standard... and you can use them to bootstrap a portable implementation of R5RS macros.

  [Guile uses Lisp-style macros fairly heavily, so Guile programmers should definitely pay attention.] ]
- [ You might need to program in Lisp some day, or talk intelligently about Lisp. ]
- [ People keep reinventing them, and not noticing that they were invented decades ago, for Lisp—I've seen at least three languages with reinventions of Lisp macros, usually in an inferior form. I want to make it clear what Lisp macros do, and what's good and bad about them, to avoid further awkward reinventions of the wheel. ]

### 7.4.1 Ultra-simple Lispish Macros

The classic macro system is the Lisp macro system, which allows the user to define an arbitrary Lisp procedure to rewrite a new construct. (Most dialects of Lisp, e.g., Common Lisp, have a macro facility of the same general kind, called `defmacro`.) We'll talk for a moment about a simplified version of Lisp-style macros. Later we'll explain why and how Scheme macros are better, at least for most purposes.

Suppose we have a macro system that we can use to tell the interpreter or compiler that when it sees an expression that's a list starting with a particular symbol, it should call a particular routine to rewrite that expression, and use the rewritten version in its place.

For the `or` example, we want to tell the compiler that if it sees an expression of the form `(or` *a* *b*`)` it should rewrite that into an expression of the form

```
(let ((temp a)
   (if temp
       temp
       b))
```

So now we want to tell the compiler how to rewrite expressions like that. Since Lisp expressions are represented as lists, we can use normal list operations to examine the expression and generate the new expression. Let's assume our system has a special form called `define-rewriter` that lets us specify a procedure of one argument to write a particular kind of expression.

Here's a rather ugly rewriter macro for `or`:

```
; OR with subtle scope bug
(define-rewriter 'or          ; tell compiler how to rewrite (or ...)
   (lambda (expr)
      (let ((a (cadr expr))
            (b (caddr expr)))
         (cons 'let                                 ; make LET form
               (cons (list (list 'temp a)))         ; make let binding form
                     (append '(if temp temp)        ; make IF form
                             (list b))
```

There's actually a scoping problem with this macro, which I'll ignore for now—it's the problem that define-syntax fixes. Later, I'll show what's wrong and fix it, but for a while I just want to talk about basic syntax of Lisp-style macros.

Now when the interpreter or compiler is about to evaluate an expression represented as a list, it will check to see if it starts with `or`. If so, it will pass the expression to the above rewriter procedure, and interpret or compile the resulting list instead.

(Actually, macroexpansion doesn't have to happen just before interpreting or compiling a particular expression. The system might rewrite all of the macro calls in a whole procedure, or a whole program, before feeding the procedure or program to the normal part of the compiler. It's easier to understand macros if they're interleaved with expression evaluation or compilation, though—it's just an extra case in the main dispatch of your interpreter or compiler.)

Implementing `define-rewriter` is easy. (We'll show an implementation for our example interpreter in a later section.) We only need to do two simple things:

- Provide a procedure that can add rewriter procedures to a table, keyed by the name of the forms they rewrite.
- Modify the interpreter (or compiler) to check whether expressions of the form (*symbol* ...) begin with the name of a rewriter macro, and if so, to call the rewriter to transform the expression before interpreting (or compiling) it.

That's all.

The above system works, but it has several awkwardnesses. One is that it is tedious to write routines that construct s-expressions directly. We can use `quasiquote` to make this easier. It will allow us to simply write the s-expression we want the macro to produce, and use unquote to fill in the parts we get from the arguments to the macro.

```
; OR with subtle scope bug
(define-rewriter 'or       ; tell compiler how to rewrite (or ...)
   (lambda (expr)
      (let ((a (cadr expr))
            (b (caddr expr)))
         `(let ((temp ,a)) ; return an s-expression of this form
             (if temp
                 temp
                 ,b))
```

This is much easier to read. The backquoted expression is now readable as code—it tells us the general structure of the code produced by the macro, and the commas indicate the parts that vary depending on the arguments passed to the macro.

Note that there is no magic here: `define-rewriter` and quasiquotation can be used independently, and are very different things. It just happens that quasiquotation is often very useful for the things you want to do in macros—returning an s-expression of a particular form.

This simple rewriting system is still rather tedious to use, for several of reasons. First, we always have to quote the name of the special form we're defining. Second, it's tedious to write a `lambda` every time. Third, it's tedious to always have to destructure the expression we're rewriting to get the parts we want to put into the expression we generate. ("Destructure" means take apart to get at the components—in this case, subexpressions.)

#### 7.4.1.1 Better Lisp-style Macros

It would be nice if the macro facility allowed you to declare the argument pattern to the macro, and automatically destructured it for you. Most Lisp systems have a special form called `defmacro` that does this for you, and avoids the need to write a `lambda` expression every time. For consistency with Scheme naming conventions, we'll call our equivalent `define-macro`.

`define-macro` implicitly creates a transformation procedure whose body is the body of the define-macro form. It also implicitly destructures the expression to be transformed, and passes the subexpressions to the transformation procedure.

Using `define-macro`, we can write `or` this way, specifying that `or` takes two arguments:

```
; OR with subtle scope bug
(define-macro (or a b)
   `(let ((temp ,a))
      (if temp
          temp
          ,b))
```

We didn't have to write the code that destructures the form into `a` and `b`—`define-macro` did that for us. We also didn't have to explicitly write a `lambda` to generate the transformation procedure; `define-macro` did that too.

This makes the syntax of `define-macro` similar to a procedure-defining `define` form. Still, you should always remember that you're not writing a normal procedure: you're writing a procedure to transform code before it is interpreted or compiled. The combination of automatic argument destructuring and template-filling (using backwuote and comma) makes it easier to do this in many cases.

Like a procedure, a macro can take a variable number of arguments, with the non-required ones automatically packaged up into a rest list. We can define a variable-arity `or` with `define-macro`:

[ need to check this example—it's off the top of my head ]

```
; variable arity OR with subtle scope bug
(define-macro (or . args)
   (if (null? args)  ; zero arg or?
       #f
       (if (null? (cdr? arg-exprs)) ; one arg or?
           (car arg-exprs)
           `(let ((temp ,(car arg-exprs)))
               (if temp
                   temp
                   (or ,@(cdr arg-exprs)))))))
```

Here we're just accepting the list of argument expressions to the `or` expression as the rest list `args`.

If it's an empty list, we return `#f`. Keep in mind that we're returning the `#f` object, which will be used in place of the `or` expression, i.e. as the literal `#f` to use in the resulting code. (Conceptually, it's a fragment of a program code, even though that program fragment will in fact return the value #f when it executes, because #f is self-evaluating. We could have quoted it to make that clearer.)

If it's a one-element list, we just return the code (s-expression) for the first argument.

If it's a list of more than one argument expression, we return an s-expression for the `let` with a nested `if`. (Note the use of `unquote-splicing` (`,@`) to splice the `cdr` of the expression list into the `or` form as its whole list of arguments.)

You should be aware, though, that what you're really doing is specifying a procedure for transforming expressions before they're compiled or interpreted, and that quasiquote is just syntactic sugar for procedural code that constructs an s-expression.

`define-macro` is easy to write, once we've got `define-rewriter`; we don't have to modify the interpreter or compiler at all. We just use `define-rewriter` to write `define-macro` as a simple macro. We'll make `define-macro` a macro that generates transformation procedures, and uses `define-rewriter` to register them with the interpreter.

### 7.4.1.2 Problems With Lisp-Style Macros

Earlier we alluded to a lurking bug in our `define-rewriter` and `define-macro` definitions for `or`.

Suppose we use the `or` macro this way—we check to see if someone is employed as either a permanent or temporary employee, and generate a w2 tax form if either of those is true.

```
(let ((temp (member x temporary-employees))
      (perm (member x permanent-employees)))
   (if (or temp perm)
       (generate-w2 x)))
```

The expansion of this is:

```
(let ((temp (member x temporary-employees))
      (perm (member x permanent-employees)))
   (if (let ((temp temp))
          (if temp
              temp
              temp)))    ;BUG! (should refer to outer temp, not inner)
       (generate-w2 x)))
```

The problem here is that we happened to use the same name, `temp`, at both the call site and inside the macro definition. The reference to `temp` in `(or temp perm)` gets "captured" by the binding of temp introduced in the macro.

This occurs because a normal macro facility does not understand issues of name binding—the name `temp` refers to one program variable at the call site, and another at the site of its use inside the macro—and the macroexpander doesn't know the difference. To the macroexpansion mechanism, the symbol `temp` is just a symbol object, not a name of anything in particular, i.e., a particular program variable.

There are two ways to get around this problem. One is for the macro-writer to be very careful to use names that are very unlikely to conflict with other names. This makes code very ugly, because of the unnatural names given to variables, but more importantly, it's harder to get right than it may seem. The other way around the problem is to get a much smarter macro facility, like the new Scheme `define-syntax` macro system.

#### 7.4.1.3 Ugly Hacks Around Name Conflicts

One way to avoid name conflicts is to pick names for variables used inside macros in such a way that they're unlikely to conflict with names that users of the macros might pick, e.g.,

```
(define-macro (or first-arg second-arg)
   `(let ((temp!in!or!macro ,first-arg)
       (if temp!in!or!macro
           temp!in!or!macro
           ,second-arg)))
```

It's unlikely that anyone will name a different variable `temp!in!or!macro` someplace else, so the problem is solved, right? Not necessarily.

Besides the fact that this is incredibly tacky, there's still a situation where this kind of solution is *likely* to fail—when people nest calls to the same macro. Each nested call will use the same name for different variables, and things can go nuts. (Food for thought: is this true of the `or` macro above? Does it nest properly?)

The standard hack around that problem is to have each *use* of the macro use a different name for each local variable that might get captured. This requires some extra machinery from the underlying system—there has to be a procedure that generates new, unique symbols, and which can be called by the macro code each time the macro is expanded. The traditional Lisp name for this procedure is `gensym`, but we'll call it `generate-symbol` for clarity.

Now we can write a fixed version of the two-argument `OR` macro.

```
; Version of 2-arg OR with scope bug fixed

(define-macro (or first-arg second-arg)
   (let ((temp-name (generate-symbol)))
      `(let ((,temp-name ,first-arg)
          (if ,temp-name
              ,temp-name
              ,second-arg)))
```

Notice that the outer `let` is outside the backquote—it will be executed when the macro is used (i.e., once each time an `or` expression is rewritten; the quasiquoted part is the code to be interpreted or compiled (after the comma'd holes are filled in).

Each time a call to `or` is processed by the compiler (or interpreter), this `let` will generate a new symbol before translating it; quasiquote will fill in the holes for the new symbol. (Be sure to get your metalevels right here: `temp-name` is the name of a variable in the macro transformation procedure, whose binding will hold a pointer to the the actual name symbol that will be used for the variable.)

Isn't this ugly? To some degree, Lisp macros are nice because you can use the same language (Lisp) in macros as you can in normal code. But due to these funky scoping issues, you effectively end up having to learn a new language—one with lots of generate-symbol calls and commas.

On the other hand, maybe it builds character and abstract reasoning abilities, because you have to think a lot about names of names and things like that. Fun, maybe, but not for the faint of heart.

## 7.5 Implementing Simple Macros and Quasiquote

### 7.5.1 Implementing Simple Macros

### 7.5.2 Implementing `quasiquote` and `unquote`

[ This section is particularly rough and needs to be reworked. Sorry. ]

`quasiquote` is a special form that (like `quote`) has a very special sugared syntax. Part of this syntax is recognized by the reader, rather than the compiler or interpreter proper; the rest of the work is done by the compiler or interpreter.

#### 7.5.2.1 Translating backquotes to `quasiquote`

A backquote character is interpreted very specially by the Scheme (or Lisp) reader, and backquoted expressions are converted into `quasiquote` expressions with a normal-looking nested-prefix-expression syntax. The expression `` `(a b c) `` is actually just shorthand for `(quasiquote (a b c))` Similarly, comma'd expressions are converted, e.g. `(a ,b ,c) is read in as `(quasiquote (a (unquote b) (unquote c)))`. Notice that as far as the reader is concerned, these are just lists—it is up to the compiler or interpreter to process them further, and the reader just preprocesses them into lists that the compiler or interpreter can deal with.

#### 7.5.2.2 `quasiquote`

The `quasiquote` special form may be built into the compiler or interpreter, but it can be implemented as a macro, in Scheme. That's the easy way to do it, and it's what we'll do.

I'll show a simplified version of `quasiquote` that only deals with commas at the top level of a list, e.g.,

```
(quasiquote (foo ,bar (baz x y)))
```

but not

```
(quasiquote (foo ,bar (baz ,x y)))
```

Notice that `(quasiquote (foo ,bar (baz x y)))` should expand to something like

```
(list 'foo bar '(baz x y))
```

We'll actually generate an expression that uses `cons` instead of `list`, because we want to write `quasiquote` recursively; if its argument is a list, it will peel one element at a time of off the list of arguments, and either quote it or not before using it in the resulting expression that is the rewritten version of the macro call.

Given this strategy, `(quasiquote (foo ,bar (baz x y)))` should expand to

```
(cons 'foo
      (cons bar
            (cons '(baz x y))
                  '()))
```

Notice that what we've done is generate an expression to generate a list whose components are explicitly quoted where necessary, as opposed to the original backquoted list where things are quoted by default and explicitly unquoted.

And since `'`*thing* is just a shorthand for `(quote` *thing*`)`, we'll really generate an ugly expression like

```
(cons (quote foo)
      (cons bar
            (cons (quote baz x y)
                  '())))
```

written as a straighforward low-level macro. We'll define it as a trivial macro that just calls a procedure `quasiquote1` to do the actual transformation.

[ NEED TO DEBUG THIS... PRW ]

```
(define-macro (quasiquote expr)
   (quasiquote1 expr))
```

```
(define (quasiquote1 expr)
   (if (not (pair? expr)) ; if quoted expr is not a list, it's just
       expr               ; a literal
       ; else we'll grab a subexpression and cons it (appropriately
       ; quoted or not) onto the result of recursively quasiquoting
       ; the remaining arguments
       (let ((first-subexpr (car expr))
             (rest-subexprs (cdr expr)))
          (if (and (pair? next-subexpr)
                   (eq? (car first-subexpr) 'unquote)))
              (list 'cons
                    first-subexpr       ; gen expr to eval this subexpr
                    (quasiquote1 rest-subexprs))
              (list 'cons
                    (list 'quote first-subexpr)    ; quote this subexpr
                    (quasiquote1 rest-subexprs)))))
```

A full implementation of quasiquote is a little trickier, because it must deal with nested uses of `quasiquote` and `unquote`; each subexpression that is not unquoted must be traversed and treated similarly to the top-level list—i.e., rather than just using the subexpressions as literals and quoting them, an equivalent expression should be constructed to create a similarly-structured list with the unquoted holes filled in. Also, a full implementation should handle `unquote-splicing` as well as `unquote`.

#### 7.5.2.3 `define-rewriter`

In Chapter [ whatever ], I showed the code for an interpretive evaluator that was designed to support macros. In this section, I'll explain how to implement the macro processor and install it in the interpreter.

Recall that when `eval` encounters an expression that's represented as a list, it must determine whether the list represents a combination (procedure call), a built-in special form, or a macro call. It calls `eval-list` to do this dispatching.

Also recall that we implemented environments that can hold different kinds of bindings—of normal variables or macros. A macro binding holds a transformation procedure that can be used to rewrite an expression before it is interpreted.

`eval-list` checks to see if the list begins with a symbol, which might be the name of a macro, or the name of a procedure. It looks in the environment to find the current binding of the symbol.

If it's a syntax (macro) binding, `eval-list` it extracts the transfromer procedure from the binding information, and calls `eval-macro-call` to evaluate the list expression.

Here's `eval-macro-call`:

```
(define (eval-macro-call transformer expr envt)
   (eval (transformer expr) envt))
```

All it does is apply the transformation procedure to the expression, and call `eval` recursively to evaluate the result.

This is sufficient to be able to *use* macros, once they're defined. We also need to be able to define macros, so that we can use them.

For that, we'll add one special form to our interpreter, `define-rewriter`, which takes a name symbol and a transformation procedure as its arguments.

[ Show `define-rewriter` ... has to accept a closure in our language, not the underlying Scheme ]

#### 7.5.2.4 define-macro

Once we've added define-rewriter to our interpreter, we don't have to modify the interpreter at all to add `define-macro`. We can simply define it as a macro in the languauge we interpret, using `define-rewriter` "from the inside." We had to add `define-rewriter` to the language implementation itself, but once that's done, we can bootstrap a better macro system with no extra help from the interpreter.

`define-macro` does three things:

It analyzes the calling form of a macro (the argument pattern) and generates code to *destructure* expressions of that form.

it creates a procedure that will do the destructuring *and* the transformation expressed in the macro body.

it installs a new syntax binding in the current binding environment, holding that transformation procedure.

Bear in mind that the following code is *not* code in the interpreter, but code to be interpreted, to create a `define-macro` macro, from inside our language.

[ show `define-macro` ... pattern matching on arg form and creating a routine to destructure and bind... ]

## 7.6 Procedural Macros vs. Template-filling Macros

## 7.7 Programming Examples Using Macros

# 8 Records and Object Orientation

## 8.1 Records

### 8.1.1 Data Abstraction

### 8.1.2 Implementing Records

## 8.2 Objects

### 8.2.1 Object Orientation

### 8.2.2 Implementing a Simple Object System

#### 8.2.2.1 Generic Functions and Dynamic Dispatch

#### 8.2.2.2 Inheritance

# 9 Other Useful Features

## 9.1 Special Forms

## 9.2 Input-Output Facilities

### 9.2.1 `read` and `write`

### 9.2.2 `display`

### 9.2.3 Ports

### 9.2.4 `with-input-\dots` Forms

## 9.3 Useful Types and Associated Procedures

### 9.3.1 Numeric Types

#### 9.3.1.1 Floating-Point Numbers

#### 9.3.1.2 Arbitrary-Precision Integers

#### 9.3.1.3 Ratios

#### 9.3.1.4 Coercions and Exactness

### 9.3.2 Vectors

### 9.3.3 Strings and Characters

# 10 `call-with-current-continuation`

`call-with-current-continuation` is a very powerful control construct, which can be used to create more conventional control constructs, like `catch` and `throw` in Lisp (or `setjmp` and `longjmp` in C), or coroutines, and many more. It is extremely powerful because it allows a program to manipulate its own control "stack" so that procedure calls and returns needn't follow the normal depth-first textual call ordering.

Recall that we said [ WHERE? ] that Scheme's equivalent of an activation stack is really a chain of partial continuations (suspension records), and this chain is known as a full continuation. And since continuations are immutable, they usually form a tree reflecting the call graph (actually, only the non-tail calls). Normally, the parts of this tree that are not in the current continuation chain are garbage, and can be garbage collected.

If you take a pointer to the current continuation, and put it in a live variable or data structure, however, then that continuation chain will remain live and not be garbage collected. That is, you can "capture" the current state of the stack.

If you keep a captured state of the stack around, and later install the pointer to it in the system's continuation register, then you can return through that continuation chain, just as though it were a normal continuation. That is, rather than returning to your caller in the normal way, you can take some old saved continuation and return into that instead!

You might wonder why anybody would want to do such a weird thing to their "stack," but there are some useful applications. One is *coroutines*. It is often convenient to structure a computation as an alternation between two different processes, perhaps one that produces items and another that consumes them. It may not be convenient to either of those processes into a subroutine that can be called once to get an item, because each process may have complex state encoded into its control structure.

(You probably wouldn't want to have to structure your program as a bunch of incremental operations that were called by successive calls to a do-next-increment routine. It may be that the program it gets its data from can't easily be structured that way, either. Each program should probably be written with its own natural control structure, each suspending its operation when it needs the other to do its thing.)

*Coroutines* allow you to structure cooperating subprograms this way, without making one subservient to (and callable piecemeal from) another.

Coroutines can be implmemented as operations on continuations. When a coroutine wants to suspend itself and activate another (co-)routine, it simply pushes a partial continuation to save its state, then captures the value of the continuation register in some way, so that it can be restored later. To resume a suspended routine, the continuation chain is restored and the top partial continuation is popped into the system state registers. Alternation between coroutines is thus accomplished by saving and restoring the routines' continuations.

Note that in this case, we can have two (or more) trees of continuations that represent the course of the computation, and that control flow can alternate back and forth between trees. Usually, computations are structured so that most of the work is done in the usual depth-first procedure calling and returning, with occasional jumps from one routine's depth-first activity to another's.

Another use of continuations is to implement `catch` and `throw`, which are roughly like setjmp and longjmp in C. The idea is that you may want to abort a computation without going through the normal nested procedure returns. In a normal stack-based lagnuage (without continuations), this is usually accomplished by storing a pointer into the stack before starting the abortable computation. If it is necessary to abort the computation, all of the activation records above the point of call can be ignored, and the stack pointer can be restored to that point, just as though all of the invocations above it had returned normally.

This can be implemented with `call-with-current-continuation` by saving the continuation at the point where an abortable computation is begun. Anywhere within that computation, that continuation may be restored (clobbering the "normal" value of the continuation register, etc.) to resume from that point.

But `call-with-current-continuation` is more powerful than coroutines or catch and throw. Not only can we escape "downward" from a computation (by popping multiple partial continuatons at once without actually returning through them), we can also escape "upwards" *back into* a computation that we bailed out of before. This can be useful in implementing *exception handling*, where we may want to transfer control to a special coroutine that may "fix up" an error that was detected, but then resume the procedure that encountered the error, after the problem is fixed.

`call-with-current-continuation` can also be used to implement *backtracking*, where the control flow backs up and re-executes from some saved continuation. In this case, we may save the continuation for some computation, but go ahead and return through it normally. Later, we may restore the saved continuation and return through it again.

Note that in general, continuations in Scheme can be used *multiple* times. The essential idea is that rather than using a stack, which dictates a depth-first call graph, Scheme allows you to view the call graph AS A GRAPH, which may contain cycles, even directed cycles (which represent bactracking).

The syntax of `call-with-current-continuation` is fairly ugly, but for some good reasons; in its raw form, it is very powerful, but correspondingly hard to use. Typically, it is encapsulated in macros or other procedures to implement other, higher-level control constructs.

`call-with-current-continuation` is itself a normal first-class procedure, which encapsulates the very low-level continuation munging abilities in something like a civilized package. Since it's a first-class procedure, you can write higher-order procedures that treat it like data, or call it, or both.

`call-with-current-continuation` is a procedure of exactly one argument, which is another procedure to execute after the current continuation has been captured. The current continuation will be passed to that procedure, which can use it (or not) as it pleases.

The captured continuation is itself packaged up as a procedure, also of one argument. That's so that you can't muck around with the continuation itself in any data-structure-like way. There are only two things you can do with captured continuations—capture them and resume them. Continuations are captured by executing `call-with-current-continuation`, which creates an *escape procedure*. They are resumed by calling the escape procedure. When called, the escape procedure abandons whatever computation is going on, restores the saved continuation, and resumes executing the saved computation at the point where `call-with-current-continuation` occurred.

Note that `call-with-current-continuation` is a procedure of one argument. We'll call that procedure the *abortable* procedure. The abortable procedure must *also* be a procedure of exactly one argument. (If you want to call a procedure that takes a bunch of arguments, and still make it abortable using `call-with-current-continuation`, you have to use a trick I'll describe below.)

The abortable procedure's argument is the escape procedure that encapsulates the captured continuation.

`call-with-current-continuation` does the following:

Creates an escape procedure that captures the current continuation. If called, this procedure will restore the continuation at the point of call to `call-with-current-continuation`.

Calls the procedure passed as its (call-with-current-continuation's) argument, handing it the escape procedure as *its* argument.

If and when the escape procedure is called, it restores the continuation captured at the point of call to `call-with-current-continuation`. We refer to this as a *nonlocal return*—from the point of view of the caller of `call-with-current-continuation`, it looks as though `call-with-current-continuation` had returned normally.

The (abortable) procedure we want to call must take one argument, which is the escape procedure that can resume the computation just beyond the call to `call-with-current-continuation`.

As if this weren't cryptic enough, the escape procedure is *also* a procedure of exactly one argument. When the escape procedure is used to perform a nonlocal return, it returns a value as the result of the call to `call-with-current-continuation`.

The argument to the escape procedure is the value that will be returned as the value of the call. Note that if the escape procedure is *not* called, and the abortable procedure returns normally, the value it returns is returned as the value of the call to `call-with-current-continuation`.

A call to `call-with-current-continuation` therefore can return in two ways. Either the abortable procedure returns normally, and `call-with-current-continuation` simply returns that value, *or* the escape procedure can be called, and its argument is returned as the value of the call to `call-with-current-continuation`.

Consider the following example, where I've given line numbers to refer to later:

```
 0: (define some-flag #f)

 1: (define (my-abortable-proc escape-proc)
 2:   (display "in my-abortable-proc")
 3:   (if some-flag
 4:       (escape-proc "ABORTED"))
 5:   (display "still in my-abortable-proc")
 6:   "NOT ABORTED")


 7: (define (my-resumable-proc)
 8:   (do-something)
 9:   (display (call-with-current-continuation my-abortable-proc))
10:   (do-some-more))

11: (my-resumable-procedure)
```

At line 11, we call `my-resumable-procedure`. It calls `do-something`, and then calls display. But before it calls display it has to evaluate its argument, which is the call to `call-with-current-continuation`.

`call-with-current-continuation` saves the continuation at that point, and packages it up as an escape procedure. (Line 9) The escape procedure, if called, will return its argument as the value of the `call-with-current-continuation` form.

That is, if the escape procedure is called, it will resume execution of the display procedure, which prints that value, and then execution will continue, calling do-some-more.

Once `call-with-current-continuation` has created the escape procedure, it calls its argument, `my-abortable-proc`, with the escape procedure as *its* argument.

`my-abortable-proc` then displays (prints) `"in my-abortable-proc."` Then it checks `some-flag`, which is false, and does *not* execute the consequent of the `if`—that is, it doesn't execute the escape procedure. It continues executing, displaying `"still inmy-abortable-proc."` It then evaluates its last body form, the string `"NOT ABORTED"`, which evaluates to itself, and returns that as the normal return value of the procedure call.

At this point, the value returned from my-abortable-proc is printed by the call to `display` in line 9.

But suppose we set `some-flag` to `#t`, instead of `#f`.

Then when control reaches line 3, the `if` *does* evaluate its consequent. This calls the escape procedure, handing it the string `"ABORTED"` as its argument. The escape procedure resumes the captured continuation, returning control to the caller of `call-with-current-continuation`, without executing lines 5 and 6.

The escape procedure returns its argument, the string `"ABORTED"` as the value of the `call-with-current-continuation` form. It restores the execution of my-resumable-proc at line 9, handing display the string `"ABORTED"` (as the value of its argument form). At this point `"ABORTED"` is displayed, and execution continues at line 10.

Often we want to use call-with-current-continuation to call some procedure that takes arguments other than an escape procedure. For example, we might have a procedure that takes two arguments besides the escape procedure, thus:

```
(define (foo x y escape)
   ...
   (if (= x 0)
       (escape 'ERROR))
   ...))
```

We can fix this by *currying* the procedure, making it a procedure of one argument.

[ An earlier chapter should have a discussion of currying! ]

Suppose we want to pass 0 and 1 as the values of x and y, as well as handing foo the escape procedure. Rather than saying

```
(call-with-current-continuation foo)
```

which doesn't pass enough arguments to the call to foo, we say

```
(call-with-current-continuation (lambda (escape) (foo 0 1 escape)))
```

The lambda expression creates a closure that does exactly what we want. It will call foo with arguments 0, 1, and the escape procedure created by `call-with-current-continuation`.

## 10.1 Implementing a Better Read-Eval-Print Loop

## 10.2 Implementing Catch and Throw

## 10.3 Implementing Backtracking

## 10.4 Implementing Coroutines

## 10.5 Implementing Cooperative Multitasking

## 10.6 Caveats about `call-with-current-continuation`

# 11 A Simple Scheme Compiler

[ The example compiler in compiler.scm is the skeleton of a simple compiler for a subset of Scheme, whose structure corresponds fairly closely to the example interpreter in eval.scm. ]

[ this is out of place, or needs more introductory intro first: ]

Where the interpreter has a basic dispatch routine called `eval`, which can evaluate any kind of expression, the compiler has a basic dispatch routine called `compile`, which can compile any kind of expression. Like `eval`, `compile` examines the expression to be compiled, and dispatches to an appropriate routine for that kind of expression. The routine that compiles an expression may recursively call compile to compile subexpressions, just as the interpretive evaluator may call `eval` recursively to evaluate subexpressions.

## 11.1 What is a Compiler?

[ This is somewhat redundant with earlier stuff, but more concrete. Should I cut it down?]

Before answering what a compiler is, let's backtrack and talk about interpreters.

### 11.1.1 What is an Interpreter?

An interpreter really does two things:

1. it examines expressions and dispatches to the appropriate code for that kind of expression
2. it performs the actual operations specified by the program

Typically, most of the work done by an interpreter is in the first category—our example interpreter, for example, spends a lot of time examining expressions to see if they're self-evaluating or symbols or lists, and dispatching to the appropriate procedure to evaluate that kind of expression. This dispatching is interleaved with the actual work that evaluates the expressions.

One of the problems with an interpreter is that every time an expression is encountered, it is analyzed again. Consider an expression like `(+ foo bar)` embedded in a loop that iterates many times. Our interpreter will encounter this expression at each iteration of the loop, and at each iteration of the loop it will do mostly the same things: it will examine the expression and find out it's a list, then call `eval-list`, which will further examine it to find out it's a combination (not a special form or macro), and call `eval-combo`. Then `eval-combo` will call `eval` recursively to evaluate the subexpressions, and each call to `eval` will examine the subexpressions and dispatch to the appropriate specialized evaluation routine. Only then do we start actually computing the value of the expression, by computing the values of the subexpressions `+`, `foo`, and `bar`, i.e., looking up the values of those variables.

### 11.1.2 OK, so what's a compiler?

We would rather factor out most of this redundant work, and examine the expression just once to see what to do. Then each time we "evaluate" the expressions, we can just do those things. For the expression `(+ foo bar)`, the set of actions an interpreter will execute (leaving out all of the analysis and dispatching is):

```
look up variable bar
look up variable foo
look up variable +
apply
```

(Here we've assumed that we evaluate subexpressions of a combination from right-to-left, rather than the more intuitive left-to-right order; that's a legal way to do it in Scheme an it turns out to be handy in a very simple compiler, as we'll explain in a minute.)

[ maybe I should change this to do args left-to-right, but the operator last, like RScheme. ]

For code like this, which doesn't have any conditionals in it, we can convert an interpreter into a compiler very easily. We just modify the interpreter so that instead of actually evaluating the expressions, it just records what operations it *would* execute if it were interpreting the expression. I'm intentionally being vague as to how exactly these simple operations (like `look-up-variable`) work, but you should be able to see that each of them should be translatable into a handful of machine instructions. That's how most compilers work: they first translate a program into an *intermediate code* representation, like our look-up-variable operations, and then translate that representation into machine instructions. (In between there may be one or more steps that "optimize" the intermediate code, and each step may represent the code in a different way.)

So this simple compiler just "pretends" to evaluate the expression, but whenever it gets to an actual action (like looking up a variable, or calling a procedure), it simply records what action it would take if it were just an interpreter. The result is a list of actions which, if taken, will have the same effect as interpreting the expression.

Here's another example:

```
(let ((x 22)
      (y 15))
   (+ x (* x y)))
```

Supposing that our intermediate code representation is a sequence of lists that represent operations and their operands, the code that our simple compiler will generate is:

```
(fetch-literal 22)
(fetch-literal 15)
(bind x y)
(look-up-variable y)
(look-up-variable x)
(look-up-variable *)
(call-procedure)
(look-up-variable x)
(call-procedure)
(unbind)
```

Later on, we'll talk in more concrete detail about where values are temporarily stored when they're looked up, and various tweaks to make it possible to translate intermediate code into smaller and faster sequences of machine instructions. For now just notice that we can string together sequences of these intermediate code operations, and if we just translate each of them into some machine instructions, we can string those sequences of machine instructions together and get a larger sequence that we can execute to evaluate the whole expression. We can execute it as many times as we want, and all of the expression analysis and dispatching will already have been done—the only work done each time it's executed is the work that actually binds variables, looks up values, calls procedures, and so on.

It's not much harder to compile conditional expressions like `if`. When we compile an `if`, we need to generate code for the condition expression, the consequent expression, and the alternative expression. (The "consequent" is the code executed if the condition is true, and the "alternative" is the code executed if it's false.) Then we need to string the code for those expressions together appropriately with some conditional branches:

```
   <code for condition>
   (branch-if-false "else-action-label")
   <code for consequent>
   (branch-unconditionally "end-label")
"else-action-label"
   <code for alternative>
"end-label"
```

The labels here will actually be translated into the addresses of the code they label, and the branches will be filled in with those addresses. (We have to be careful to use a unique pair of new labels for each `if` we compile, so or some other trick like that, so that we can nest `if` expressions and keep their labels straight.)

(One way of generating machine code from this representation is to translate each of the statements into a short sequence of assembly instructions and each label into an assembly label, stringing them together as shown. Then the assembly code can be assembled into machine code.)

Note that for an `if`, the control structure of the compiler is actually simpler than the control structure of an interpreter. The interpreter will evaluate the condition expression, and then decide at run time whether to evaluate the consequent ("then") expression or the alternative ("else") expression. The compiler will always compile all three subexpressions, and string them together with conditional branches that will do the deciding at run time, based on the runtime value computed by the code for the condition expression.

Here's a slightly more complicated example:

```
(let ((x 15))
   (if x
       (let ((y (* 2 x)))
          (+ x y))
       #t))
```

translates to intermediate code roughly like:

```
    (fetch-literal 15)
    (bind x)
    (lookup-variable x)
    (branch-if-false "else22")
    (lookup-variable x)
    (fetch-literal 2)
    (lookup-variable *)
    (call-procedure)
    (bind y)               ; create and enter envt that binds y
    (lookup-variable y)
    (lookup-variable x)
    (lookup-variable +)
    (call-procedure)
    (unbind)               ; exit envt that binds y
    (branch "end22")
  "else22"
    (fetch-literal #t)
  "end22"
```

There are actually a couple of minor things wrong with the code we've generated, but this is pretty close to a workable intermediate representation.

## 11.2 What Does a Compiler Generate?

[ Talk about machine code, interpretive virtual machines, etc. ]

## 11.3 Basic Structure of the Compiler

The main function of the compiler is `compile`, which generates intermediate code for an expression, and which may call itself recursively to generate code for subexpressions.

Calls to `compile` hand it an expression and some bookkeeping information we'll discuss later. Compile returns intermediate code, plus updated versions of some of the bookkeeping information.

To start this process off, top-level forms (like the ones you type into the `read-compile-run-print` loop, or definitions of top-level procedures) are massaged a little, then intermediate code for them is generated. Then the intermediate code is converted into real executable code and packaged up as a closure that can be called.

We will discuss these issues of massaging top-level forms and generating executable closures later; for now, the main thing to understand is the recursive generation of intermediate code for nested expressions.

Here's the main dispatch routine of the compiler, which is analagous to the interpreter's `eval`:

```
(define (compile expr c-t-envt literal-state c-t-cont)
   (cond ((symbol? expr)
          (compile-symbol expr c-t-envt literal-state c-t-cont))
         ((pair? expr)
          (compile-list expr c-t-envt literal-state c-t-cont))
         ((self-evaluating? expr)
          (compile-self-eval expr c-t-envt literal-state c-t-cont))
         (#t
          (syntax-error "Illegal expression form" expr))))
```

For now, ignore most of the arguments to `compile`, we'll explain them later. The main thing to notice is that it looks a lot like `eval`.

[ blah blah blah...]

[ Somewhere, it's important to bring out the difference between the mutual recursion of eval and apply in the interpreter and the way the compiler works. Eval recurs locally, but just generates code for apply... The control structure of the compiler is actually simpler than for the interpreter, because the hairy stuff just happens at run time... ]

## 11.4 Data Representations, Calling Convention, etc.

Before trying to understand the compiler itself in more detail, it is probably best to have a concrete idea of what the representations of data are, how procedure calls work, and how registers are used. That is, you have to understand the "abstract machine" that the compiler compiles for.

An abstract machine is an abstraction of low-level operations and objects. The compiler first compiles code from the source language into this lower-level representation, and then translates the "abstract machine language" into actual executable code. (The executable code may be machine code that runs directly on the hardware, or an interpretive executable code such as bytecodes, which are interpreted by a fast interpreter.)

You can think of an abstract machine as being more like an assembler than an interpreter, but maybe a little smarter than most assemblers.

I will describe one particular set of features to make things concrete; this is not quite how RScheme works, or Scheme-48, or any particular other system that I know of, but there's nothing unusual about it except maybe its simplicity.

In fleshing out our example compiler, let's suppose our system works this way:

1. We have several important registers used in stereotyped ways, e.g., to hold a pointer to the current binding environment.
2. We have an evaluation stack that's used to store intermediate values while evaluating nested expressions.
3. We use a continuation chain to represent the saved state of callers, their callers, and so on, so that they can be resumed after a procedure returns.

The registers of the abstract machine may represent hardware registers, or just storage locations that are used in these stereotyped ways. (For example, if compiling to C, the registers might be C global variables, and the C compiler might or might not let you specify that the variables should be put in hardware registers.)

### 11.4.1 The Registers

We'll assume that there are several important registers that can be used by the code our compiler generates:

1. The `VALUE` register, where an expression leaves a value so that it can be used by an enclosing expression. In The case of a procedure, this is where the value is left for the caller when the procedure returns. The value register is also used when calling a procedure.
2. The `ENVT` register, which holds a pointer to the environment that code is currently executing in.
3. The `CONT` register, which holds a pointer to the chain of saved continuations of callers.
4. The `TEMPLATE` register, which holds a pointer to a special data structure associated with the procedure that is currently executing, and

5. The `PC` (program counter) register, which says which instruction we are currently executing. (If we're compiling to normal machine code, this is a special register built into the CPU for this purpose, and we use it pretty much like any other code would. If we're compiling to an interpretive executable code, this is probably variable in the interpreter.)

### 11.4.2 The Evaluation Stack (or Eval Stack, for short)

The eval stack is used for holding values that have been computed by evaluating subexpressions, but not yet used or bound.

In evaluating the expression `(+ foo 22)`, the three values will be computed. When each value is computed, it will be left in the `VALUE` register. We evaluate right to left, and after evaluating each argument, we perform a `PUSH` operation on the eval stack, which copies the value in the value register onto the eval stack. When we get to the first subexpression (the one that's supposed to return a function to call), we leave the value in the value register, because that's where we put the closure pointer for a procedure call.

The eval stack is used for two main purposes:

1. storing intermediate values for nested expressions, and
2. passing arguments to procedures.

The eval stack is *not* used to hold intermediate values or local variables for suspended procedures—it isn't like the activation stack in a conventional implementation of C or Pascal. The values in the eval stack at any given time are only the intermediate values stored for the *currently executing* procedure. Intermediate values for suspended procedures are saved in the continuation chain as necessary.

When we call a procedure, the only values on the eval stack are the arguments to the procedure. Any other values used by the caller are moved from the eval stack into a continuation before calling.

### 11.4.3 The Continuation Chain

The continuation chain is a data structure that fills roughly the role of an activation stack in the implementation of a conventional programming language. The continuation chain is a linked list of *partial continuations*, each of which is a record that stores the saved state of a procedure.

When a procedure performs a non-tail procedure call, it packages its important state information up into a partial continuation; this record saves the values of the environment, template, PC, and continuation registers, and any temporary values on the eval stack.

Once a caller has saved its state in a partial continuation, then the callee can do whatever it wants with the important registers and the evaluation stack. (This is called a *caller-saves* register usage convention, because the caller of a procedure is obliged to save any values that it will need when it resumes.)

Remember that continuations are allocated on the garbage collected heap and are immutable—we never modify a continuation once it's created. When we resume from a saved partial continuation, we copy the values from the partail continuation into the registers and eval stack, but that doesn't modify the partial continuation itself—it's still sitting out there on the heap. This is important for being able to implement call-with-current-continuation: it's what allows us to resume from the same continuation any number of times.

### 11.4.4 Environments

The compiler assumes that a binding environment is a chain of *frames*, each of which is a vector of slots which are the variable bindings. Each frame also has a *static link* or *scope link* field, which points to the frame representing the next lexically enclosing environment.

Top-level environments are represented specially, as hash tables that map names to bindings. We'll use a hash table instead of the association lists we used in our simple interpreter, because they're faster if you have a lot of bindings. A binding object for a top-level environment is pretty much the same as in the interpreter: a little vector with two important slots: a slot for its name and another slot that is the actual value field.

Local environments are represented very differently from toplevel environments: each frame is a vector of slots, and does *not* store the names of the bindings. It turns out that the names are only needed at compile time, so they don't actually have to be stored in the runtime environment. (The compiler also turns out to be able to do most of the work for looking up a toplevel variable at compile time, so the speed of our hash tables is not going to be critical to our runtime speed.)

### 11.4.5 Closure Representation and Calling

Closures in our system are represented as objects with two fields: a pointer to the environment captured by the closure, and a pointer to an object called a *template*, which in turn contains a pointer to the code for the procedure.

When we evaluate the following expression

```
(let ((foo 22)
      (bar 15))
   (lambda (...)
       ...))
```

we'll create an environment frame to hold the bindings of foo and bar, and initialize them to `22` and `15`, respectively. This environment frame will have a scope link pointing to the environment we were executing in when we entered the `let`. Inside this environment, we'll create a closure. The closure will hold a pointer to the new environment, and a pointer to a template object representing the anonymous procedure being closed. The template will have a pointer to the actual executable code. All of these things will be heap-allocated objects, and in our implementation we'll give each one header field showing what kind of object it is:

```
                                          <to envt. frame
                                           for enclosing
                                              scope>

                                                ^
                                                |
                                +----------+    |
                                | envt. fr.|    |
                        ,------>+==========+    |
                       /  scope |     +----+----+
     +----------+     /         +----------+
     | closure  |    /      foo |       22 |
     +==========+   /           +----------+
envt |     +----+--'        bar |       15 |
     +----------+               +----------+
proc |     +----+--.
     +----------+   \      +----------+
                     \     | template |       +----------+
                      `--->+==========+       |   code   |
                      code |     +----+-----> +==========+
                           +----------+       |executable|
                           |          |       + code for +
                           +----------+       |procedure |
                           |          |       +          +
                           +----------+       |   ...    |
                           |   ...    |       +----------+
                           +----------+
```

The template object holds not only the pointer to the actual code, but any other handy values that the compiler can compute or look up at compile time, and which should be available to the procedure at run time. We'll discuss that more later.

When we want to apply a procedure to some argument values, we put the argument values on the eval stack, and a pointer to the closure we want to call in the `VALUE` register. Then we execute a short sequence of instructions that does the call:

Extract the environment pointer from the closure and put it in the environment register. (This is basically just an indexed load using the value register as a base.)

Extract the template pointer from the closure and put it in the template register. (This is basically just another indexed load using the value register as a base.)

Extract the code pointer from the template and put it in the program counter register, i.e., jump to that code. (This is basically just another indexed load using the template register as a base, and a jump to that address.)

Thus actual machine code for our "apply" operation in our intermediate representation is just a handful of instructions that do this stuff—a stereotyped little instruction sequence that destructures a closure and puts the appropriate values in registers before beginning execution of the procedure.

Because this is the way the procedure calling convention works, we know that when we begin executing the code for a procedure, the environment register will point to the right environment (where the procedure was defined) and the template register will point to the template for that procedure. Any values stored in the template by the compiler can be fetched at compile time by doing an indexed load with the template register as a base.

Consider this procedure:

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

Here, the literal bar is needed by the procedure—there must be some code in foo that will somehow fetch a pointer to the symbol bar. That's what the template object is for. When this procedure is compiled, the compiler accumulates a list of such literals, and when the template object for the procedure is created, all of those values will be stored into it. When the compiler generates code to fetch the symbol `bar`, it just looks at the symbol's position in the literal list (and thus its offset in the template object), and generates code to do an indexed load to fetch that value from the template at run time.

## 11.5 Continuations

### 11.5.1 Applying a Procedure Doesn't Save the Caller's State

Remember that when we do a procedure call, we do *not* necessarily save the state of the caller. For a non-tail call, the compiler must generate code to save the caller's state plus code for the actual call. For a tail call, there is no need to save the state. Because of this, there isn't really a single "procedure call" operation that saves the caller's state and invokes the callee. There are two separate operations, `save-continuation` and `apply`.

As mentioned above, the code sequence that performs a procedure applicatin assumes that the pointer to the closure to be called is in the `VALUE` register. The procedure will leave its value in that register when it returns.

### 11.5.2 Continuation Saving

`save-continuation` is the operation that saves the state of the currently executing procedure in a partial continuation, and pushes it onto the continuation chain.

When pushing a continuations, it is important to save all of the values on the eval stack, *except* for the arguments to the procedure being called. Therefore, when generating code for a combination (procedure call) expression, the code to save the caller's state does *not* come just before the actual code to call the procedure. This would remove the arguments to the procedure from the eval stack. Instead, the continuation save comes just before the code that generates the argument values that will be passed to the procedure:

```
  (save-continuation <label>) ; save everything else before computing args
  <compute argn>
    ...
  <compute arg1>
  <compute callee>
  (apply)
<label>
```

that way, the arguments to the call (and nothing else) will be on the eval stack when the procedure is called, and when the procedure returns, it will restore the other values from the partial continuation onto the eval stack.

This separation of the saving and calling code looks especially funny for nested expressions that call procedures, but it makes perfect sense.

`save-continuation` takes an argument which is the address of the code to execute when the continuation is resumed. This address is saved in the partial continuation, and when the continuation is resumed, it will be branched to (put in the `PC` register).

### 11.5.3 An Example

Now that we have a more detailed idea how the registers, eval stack, and continuations work, here's an example:

```
(+ (- a b) (/ c d))
```

compiles to intermediate code something like:

```
  (push-continuation "resume23") ; save cont for call to +
  (push-continuation "resume24") ; save cont for call to -
  (lookup-variable d)   ; get value of d into value reg
  (push)                ; push value of d on eval stack
  (lookup-variable c)   ; get value of c into value reg
  (push)                ; push value of c on eval stack
  (lookup-variable /)   ; look up /
  (apply)               ; call /, which is in value reg after lookup
  (push)
"resume24"
  (push-continuation "resume25") ;save cont for -, incl. value of (/ c d)
  (lookup-variable b)   ; get value of b
  (push)
  (lookup-variable a)   ; get value of a
  (push)
  (lookup-variable -)   ; get value of -
  (apply)               ; call -
  (push)                ; push returned value on top of restored e stack
"resume25"
  (lookup-variable +)   ; look up +
  (apply)               ; tail call +
"resume23"
```

Things to notice:

1. after the first `apply`, the called routine (or something it directly or indirectly tail calls) will eventually do a procedure return, and pop the latest continuation we pushed, restoring anything that was on the eval stack at that point, and resuming execution at label1. [ OOPS... fix this ]
2. after the second apply, the called routine will eventually (directly or indirectly) do a procedure

return, which will pop the second continuation we pushed, restoring the already-computed value of the subexpression `(/ c d)` to the eval stack.

3. we generated code for the expression `(+ (- a b) (/ c d))` to be used in tail position. This code doesn't save a continuation before the final call to `+`. If the expression is to be used in non-tail position, we must generate slightly different code, which will save a continuation that will resume after this expression.

### 11.5.4 Generating Unique Labels

[ where does this go? ]

Like `compile-if`, `compile-combo` generates labels as necessary to be able to name the code where execution should be resumed after a call—in the code it generates, it puts the label just before the intermediate code instruction to resume, and the same label in the call to `save-continuation` that should resume there.

It is easy to generate unique labels for every resumable point in a program. We just keep a counter of labels we've used so far, and to create a new one we append the digits representing this number to the string `"resume"`, so that we get `"resume1"`, `"resume2"`, and so on.

We can write a Scheme procedure, `generate-label`, which keeps a counter, and when given a string as an argument, returns the a new string with the same characters plus the digits representing the number in the counter. That way, we can use labels that start with `"else"` and `"end"` to label the branch targets of an `if` expression, and labels that start with `"resume"` to represent the resumption points for continuation saving. This makes the intermediate code we generate fairly understandable, while ensuring that labels are still unique, and easy to use as assembler labels when translating intermediate code to machine language.

## 11.6 More on Representations of Environments

To get reasonable performance for our system, we'll want to treat the top-level environment differently from local variable binding environments. We'll use a trick involving lexical scope to precompute most of the work done in looking up a local variable binding, and a different trick to make it fast to look up top-level variables.

As we said before, each local variable binding contour (e.g., the bindings introduced by a `let`, or by binding the args to a procedure) is represented at run time as a frame with slots for each variable, plus a scope link that points to the frame representing the enclosing contour.

A top-level environment is likely to be large, and we will want to be able to access it in special ways. We will represent it as a hash table that maps symbols (variable names) to their toplevel bindings. The bindings themselves will be represented as objects, whose main function is to have one field that holds the current value of the variable. For simplicity, we'll make them self-identifying as well: not only will the names be used as keys in the hash table, but the binding objects will hold pointers to their names as well as values.

Keep in mind that this representation is just one that's convenient. A toplevel environment could be represented as any kind of table (e.g., an association list), but we want it to be reasonably fast to access even if there are thousands of top-level variables. (We'll use a special trick to make normal accesses to top-level variable bindings very fast at run time, but we want to make them reasonably fast at compile time as well, and hash tables are good for that.)

Suppose we evaluate the following expressions at the top level, to define and initialize a couple of variables:

```
(define quux "fubar")

(define (double x) (+ x x))
```

This will modify the toplevel environment by adding bindings for `quux` and `double`, in addition to what's already there:

```
 +--------------------------------------------------------------------+
 |                                                                    |
 |                                                                    |
\|/                                                                   |
 +------------------+                                                 |
 |   toplevel env.  |                                                 |
 +==================+         +----------+                            |
 |   cons |    *----+-------->| binding  |                            |
 +--------+---------+         +==========+                            |
 |        |         |   value |     *----+------><closure for cons>   |
      .        .              +----------+                            |
      .        .         name |     cons |                            |
      .        .              +----------+                            |
                                                                      |
 |        |         |         +----------+                            |
 +--------+---------+         | binding  |                            |
 |   quux |    *----+-------->+==========+                            |
 +--------+---------+   value |     *----+------>"fubar"              |
 |        |         |         +----------+                            |
      .        .         name |     quux |                            |
      .        .              +----------+                            |
      .        .                                                      |
                                                                      |
 |        |         |         +----------+                            |
 +--------+---------+         | binding  |           +----------+     |
 | double |    *----+-------->+==========+           | closure  |     |
 +--------+---------+   value |     *----+---------->+==========+     |
 |        |         |         +----------+      envt |     *----+-----+
      .        .         name | double   |           +----------+
      .        .              +----------+      proc |     *----+---> ...
      .        .                                     +----------+
```

Several things to note:

The representation of the hash table itself may not really be a simple array of name-value pairs, but I didn't want to clutter up the picture with overflow buckets and whatnot.

In principle, we don't need to have pointers to separate binding objects. We could just store the values of bindings right in the table, using the value fields of the name-value pair to hold the actual variable values. (After all, a binding is really just a location with a name, used to hold a value.) It will turn out to be convenient for our implementation to have separate objects that hold the values.

The occurrences of symbol names in the picture would really be pointers to symbol objects, and the string `"fubar"` would really be an object itself as well. As usual, we selectively abbreviate our pictorial representation to avoid cluttering things up.

We refer to the toplevel binding objects as objects, but they're not Scheme objects—standard Scheme doesn't give you any way to get a pointer to one of these and play with it from inside the language. These "objects" are objects in the sense that they're allocated on the heap and referred to via pointers by the compiler and by compiled code, but they're not "first class." (An extended version of the Scheme language may `let` you get at them from inside the language, but that's not standard.)

## 11.7 Compiling Code for Literals

When the compiler compiles code for a literal, like `'foo` or `"foo"` or `22` in the following expression,

```
(list 'foo "foo" 22)
```

it notices which literals the expression will need at run time, and ensures that those literals will appear in the template of the procedure. It keeps a list of literals needed by the procedure it's compiling, and after compiling the code for the procedure, it uses that list to construct the template that goes with the code.

If `foo` is the first literal encountered, it might be put into the list first, and assigned the first free slot in the template (after the code pointer). `"foo"` might be assigned the second slot, and so on. In the terminology of language implementation, the template acts as a *literal frame*, as well as holding the pointer to the procedure's code.

After assigning a literal a position in the template, the compiler can generate one or two instructions that can fetch the value out of the template, by using the address of the template, adding the offset of that slot, and loading from the resulting address. Since the template address is guaranteed to be in the `TEMPLATE` register, this is probably just a single indexed load instruction. In pseudo-C, it might look like:

```
value_register = *(template_register + offset)
```

where offset is computed by the compiler and therefore is probably an immediate operand to the load instruction that loads the value into the value register.

Notice that here we're taking advantage of the fact that the compiler runs in our system, and generates code that's just data in our system. The code will run in the same heap, and the compiler can therefore just compute values and put them in the template, and they'll stay around until that code is executed. (Life gets a little more complicated if you want to generate code that will be loaded into a different system and executed there.)

[ Now we should explain that the `literal-state` argument to `compile` is the list of literals seen so far in compiling a procedure. The return value of `compile` is intermediate code that includes an updated `literal-state`. ]

## 11.8 Compiling Code for Top-Level Variable References

Because of lexical scoping, it is easy for the compiler to tell the difference between a reference to a top-level variable binding and a reference to a local variable. We'll talk about that in detail later, but for now just assume that the compiler knows the difference at the point where it generates code for a variable reference.

When the compiler generates code for a top-level variable, it can usually look up the binding of that variable in the environment that the code is being generated for—the expression that defines the variable has already been executed, so the binding already exists.

The compiler can therefore do the actual lookup at compile time, e.g., hashing into the hash-table that implements a toplevel environment and getting a pointer to the actual binding object that will be referenced at run time.

To make references to this object fast, the compiler can simply put this pointer in the template for the procedure being compiled, as though it were a literal value.

Be clear on what's going on here: the compiler can't look up the *value* of the variable, because that's not known until the moment the variable is referenced at run time. (Before the code is executed, some other piece of code might run and change the value stored in the binding.) But the identity of the binding itself is known, and can be stashed in the literal frame.

Actually, it's just slightly more complicated than this. A variable can be used in a procedure definition before the variable itself is defined. (This is called a "forward reference.") To get around this, the compiler "cheats," and goes ahead and creates the binding object and its entry in the toplevel environment before the definition of the variable is actually encountered. At the language level, the variable hasn't been bound or given a value, but we go ahead and create the unique binding object and use it in compiling other expressions. For error checking, we put a special value in the binding to indicate that the binding isn't "real" yet—we put a reference to some object we consider "not a real value," so that we can detect uses of a variable that isn't really bound.

(In a system designed for maximum safety and early error checking, we could ensure that each reference to a toplevel variable would check for this value, and signal an error if it's found. If we're not quite so concerned with early error checking, we can wait until somebody attempts to use such a value, e.g., by adding it to something, or taking the `car` of it, and we rely on the normal type checking of the language to tell us something's wrong at the point that operation occurs.)

Now consider compiling a procedure like

```
(define (make-foo-list)
   (list 'foo "foo"))
```

The compiler will accumulate a list of toplevel bindings and literals needed for the procedure, namely a string `"foo"`, the symbol `foo`, and toplevel binding of the symbol `list`. These will be put in the template for the procedure, in the first, second, and third slots after the code pointer. The code generated for this procedure (assuming right-to-left evaluation) will be something like:

```
(fetch-literal 1) ; get string "foo" from template slot 1
(push)            ; push it on eval stack
(fetch-literal 2) ; get symbol foo from template slot 2
(push)            ; push it on eval stack
(fetch-literal 3) ; get toplevel binding of list from template slot 3
(t-l-bdg-get)     ; extract value from binding
(apply)           ; (tail-)call list
```

Notice, of course, that we've made our intermediate code representation more concrete now—we use slot numbers as the arguments to fetch-literal, and we explicitly get the value of the toplevel variable from the toplevel binding object in the value register. For setting the value in a binding, we'll use a different intermediate code instruction, `t-l-bdg-set!` (`t-l-binding-set!` expects the value register to hold a pointer to a toplevel binding object; it extracts the value of the binding, and leaves that value in the value register.)

[ Now we can explain more about literal states—we accumulate a list of literal values and top-level variable bindings that must be accessible when the procedure runs. ]

By now it should be very clear how you would translate each of these little operations in our intermediate representation into a few assembly-language instructions.

[ need picture? ]

## 11.9 Precomputing Local Variable Lookups using Lexical Scope

We can't really look up local variable bindings at compile time the way we can toplevel bindings—local variable bindings don't exist yet when we're compiling the expressions that create and use them. (Consider the fact that every time you enter a `let`, or call a procedure that binds arguments, a new binding environment frame is created. Code that executes in such environments will see a different binding environment frame in the environment register each time it runs.)

What we *can* do is take advantage of lexical scope to precompute most of the *search* for a variable in an environment.

Consider the execution of this procedure:

```
(lambda (x y)
   (let ((a <some-expression>
         (b <some-expression>))
      (list a b x y)))
```

When we compute the arguments to the call to `list`, it's obvious that the first and second variables (`a` and `b`) will be in the first and second slots of the first binding environment frame, pointed directly to by the `ENVT` register. This is the environment created by the `let`. The third and fourth variables (`x` and `y`) will be in the next environment frame, pointed to by the scope link of the first.

The compiler can compute the *lexical address* of each variable binding at the point where a reference to it is's compiled—that's the *relative location* of the variable starting from the environment register. A lexical address has two parts: the number of environment frames to skip to find the right frame, and the offset of the binding in that frame. In the above example, the lexical addresses are:

```
a:  0,1
b:  0,2
x:  1,1
y:  1,2
```

(We use the convention that frame numbers start at zero, but slot numbers appear to start at 1 because the scope link is in slot 0.)

The code generated for the reference to a can simply be an indexed load instruction, using the environment register plus an offset to grab the value in the first variable binding slot. In pseudo-C, that's something like

```
value_register = *(envt_register + offset)
```

where offset is probably 4 (bytes) to index past the scope link slot. Slightly more abstractly, its lexical address is [ WHAT? ]

The code for the reference to the variable `y` would involve one level of indirection—first the scope link pointer must be extracted from the first environment frame, and then that can be used for an indexed load to get the value of the second slot of the second frame:

```
value_register = *(envt_register)  ; get ptr to 2nd envt frame
value_register = *(envt_register + offset)
```

where offset is probably 8 (bytes) to index past the scope link and the binding of `x`.

Given this scheme, accessing a local variable takes time proportional to the number of environment frames intervening between between the expression being compiled and the environment where the referenced variable is defined. That's usually fairly fast, for two reasons:

1. The depth of lexical nesting is usually small—it corresponds to the nesting of binding expressions in the program, and is usually between one and three, an only rarely greater than five or so.
2. Most references that are executed at run time are to variables in the current scope, or maybe a level or two back from that. (Consider references to variables in inner loops, which constitute the most frequently-executed code in most programs.)

For these reasons, most references to local variables will take between one and five instructions. To a first approximation, the time to reference local variables can be regarded as a small constant. (A slightly snazzier compiler can reduce this by using more registers, and leaving many values in registers instead of pushing and popping them from the eval stack, but that's a more advanced technique than we want to address here.)

### 11.9.1 Lexical Addressing and Compile-Time Environments

Computing lexical addresses is very easy for the compiler. All it needs to do is maintain a data structure called a *compile-time environment*, which records the *structure* of the runtime environment.

Each time the compiler compiles an expression that creates new bindings, it extends the compile-time environment to reflect the change to the environment structure, and when compiling expressions that will execute in that environment, it hands the new compile-time environment to the recursive call to `compile` which will compile that expression.

For example, when compiling a `let`, the compiler dispatches to `compile-let`, the analogue of `eval-let`, which does four things:

1. Compiles code for the initial value expressions. This code executes in the environment outside the `let`, so the `compile-let` uses the environment is was given when making recursive calls to `compile` to generate the initial value code.
2. Generates code to create a binding environment and intialize it with those values.
3. Extends the compile-time environment with a new frame, reflecting the fact that the body of the `let` will execute in a new scope including the new bindings.
4. Calls `compile-sequence` to compile the body of the `let`, passing it the new compile-time environment.

Just as the overall recursive structure of the compiler closely resembles the recursive structure of the interpreter, the role of the compile-time environment is very much like the role of the environment in the interpreter.

When the interpreter (compiler) evaluates (compiles) subexpressions that execute in the same environment as their parent expressions, it hands the recursive invocation the same environment it was given. When the interpreter (compiler) evaluates (compiles) an expression in a new environment, it hands the recursive call the new (compile-time) environment.

The structure of the compile time environment at any point in the compilation process mirrors the structure of the runtime environment where the code will execute. Unlike the interpreter's representation of the environment, however, the compile-time environment doesn't contain actual bindings—it can't, and it doesn't need to.

In effect, we split the interpreter's environment into two parts with parallel structure. Where the interpreter's environements are chains of frames holding name-binding pairs, the compiler splits those into two chains of frames: the runtime environment, whose frames hold the actual bindings, and the compile-time environment, whose frames hold the corresponding names.

Consider the expression

```
(let ((x 1)
      (y 2))
   (let ((foo 3)
         (bar 4))
      (list foo bar x y)))
```

At the point where `(list a b x y)` is compiled or executed, the environment for an interpreted system appears as shown on the left, below, while the environments for a compiled system appear as shown on the right:

```
    INTERPRETED                   COMPILED              COMPILED
                               (compile time)          (run time)
                   /\                         /\                   /\
                    |                          |                    |
+---------------+   |           +----------+   |     +----------+   |
|   envt. frame |   |           | c.t.e.fr.|   |     | envt. fr.|   |
+===============+  /            +==========+  /      +==========+  /
|          +----+-'             |     +----+-'       |     +----+-'
+-------+-------+               +----------+         +----------+
|     x |     1 |               |        x |         |        1 |
+-------+-------+               +----------+         +----------+
|     y |     2 |               |        y |         |        2 |
+-------+-------+               +----------+         +----------+
                 -                         -                    -
                |\                        |\                   |\
                  \                         \                    \
                   \                         \                    \
+---------------+   |           +----------+  |      +----------+  |
|   envt. frame |   |           | c.t.e.fr.|  |      | envt. fr.|  |
+===============+  /            +==========+  /      +==========+  /
|          +----+-'             |     +----+-'       |     +----+-'
+-------+-------+               +----------+         +----------+
|   foo |     3 |               |      foo |         |        3 |
+---------------+               +----------+         +----------+
|   bar |     4 |               |      bar |         |        4 |
+-------+-------+               +----------+         +----------+
```

Note that there is a many-to-one relationship between the compile-time environments and the run-time environments: a `let` or `lambda` expression is compiled once, and the corresponding environment frame is created and passed to the recursive calls that compile subexpressions. The code may be executed many times, however, and each time a run-time environment frame will be created so that the code for subexpressions can be executed in that environment.

### 11.9.2 A Detailed Example

Taking it step by step, let's look at the compilation of the expression

```
(let ((x 1234)
      (y 3456))
   (let ((foo z))
      (+ (- foo x)
         (+ bar y))))
```

goes as follows. (We'll assume that this expression occurs at the top level.)

Since we're compiling a top-level expression, we compile it in a compile-time environment that corresponds to the top-level environment. Toplevel environments are treated specially, because they're not created dynamically the way local binding environments. There's a one-to-one relationship bewteen top-level compile-time environments and the corresponding run-time environments. We'll represent the top-level compile-time environment as a special kind of environment frame, which really just holds a pointer to the top-level runtime environment so that top-level variables can be looked up.

So we start in a top-level environment, which we'll depict as `[top-level]`; we hand this to `compile` along with the expression to compile. `compile` is the main dispatch that analyzes the expression; in this case, it analyzes it and dispatches (via `compile-list` and `compile-special-form`) to `compile-let`, the specialized procedure for compiling `let` expressions.

`compile-let` compiles the initial value expressions for `x` and `y` using `compile-multi`, which in turn calls `compile` recursively; they are compiled in the (top-level) environment, which is just passed along because no new environments have been created yet. In this case it doesn't matter, though, because they're just literals. (The values `1234` and `3456` get added to the literal list at this point.) Then `compile-let` generates the code to bind `x` and `y`.

So far, the code generated looks like:

```
(fetch-literal #1)    ; fetch 1234
(push)                ; push it on eval stack
(fetch-literal #2)    ; fetch 3456
(push)                ; push it on eval stack
(bind 2)              ; bind 2 vars (x and y), w/values form eval stack
```

and the literals list holds `1234` and `3456`.

`compile-let` then calls `compile-sequence` to compile the body of the `let`, but first it creates a new compile-time environment, to represent the fact that the body sequence will execute in the new runtime environment after `x` and `y` have been bound. The structure of this environment is

```
[ x y ] -> [toplevel]
```

(This is our new, terse way of drawing the box-and-arrow data structure for compile time environments. I got tired of drawing ascii art.)

`compile-sequence` calls `compile` recursively to evaluate a sequence of expressions; in this case, there's only one expression in the body.

The recursive call to `compile` dispatches (again via `compile-list` and `compile-special-form` to `compile-let`, to compile the inner `let`.

`compile-let` compiles the initial value expressions using `compile-multi`. `compile-multi` calls `compile` recursively to compile the one expression in the list, the symbol `z`. (Again, the same environment is just passed along, because we haven't created a new environment.)

The recursive call to compile now dispatches to `compile-symbol`, which looks up the binding information for the symbol `z` in the compile-time environment. There's no binding in the first frame (containing `x` and `y`), so the search goes to the second frame, which is the top-level environment, and the top-level binding is returned. This causes a dispatch to `compile-toplevel-var-ref`, which adds the toplevel binding of `z` to the literals list and generates code to get it from the template and extract its value at run time.

Then `compile-let` generates code to bind the fetched value as the local variable foo.

The code generated so far is:

```
(fetch-literal 1)    ; fetch 1234
(push)
(fetch-literal 2)    ; fetch 3456
(push)
(bind 2)             ; bind 2 values (x and y)
(fetch-literal 3)       ; get toplevel binding (of z)
(t-l-bdg-get)           ; get value from (z's) binding
(push)
(bind 1)                ; bind one variable (foo)
```

and the literals list contains `1234`, `3456`, and the binding of `z`.

Now `compile-let` creates a new compile-time environment to represent the environment created by the inner `let`; its structure is

```
[ foo ] -> [ x y ] -> [toplevel]
```

and it passes this to `compile-sequence` to compile the body of the let. `compile-sequence` calls `compile` recursively once, handing it the new environment, to compile the one body expression, `(+ (- foo x) (+ bar y))`.

The recursive call to `compile` dispatches (through `compile-list`) to `compile-combo`, which recursively calls `compile` three times, to generate code for the subexpressions `(+ bar y)`, `(- foo x)`, and `+`. Since no new bindings are being created, the recursive calls are given the same environment.

The recursive call to call `(+ bar y)` similarly dispatches to `compile-combo` and compiles `y`, `bar`, and `+`. Each of these calls dispatches to `compile-symbol` and the variables are looked up in the compile-time environment. The lookup for `y` returns a lexical address of 1,2, so the intermediate code generated is

```
(local-var-ref 1 2)
```

The lookup for bar doesn't find any local bindings and returns the toplevel binding so the binding is added to the literal list and the intermediate code is

```
(literal-lookup 4)
(t-l-bdg-get)
```

Similarly, the lookup for `+` doesn't find any local bindings and returns the toplevel binding, so the binding is added to the literal list and the intermediate code is

```
(literal-lookup 4)
(t-l-bdg-get)
```

now the call to `compile-combo` that compiles `(+ bar y)` can string these three fragments together to get

```
    (save-continuation "resume26") ; save state for call to +
    (local-var-ref 1 2)            ; look up y
    (push)
    (literal-lookup 4)             ; get toplevel binding (of bar)
    (t-l-bdg-get)                  ; get value from bdg (of bar)
    (push)
    (literal-lookup 5)             ; get toplevel binding (of +)
    (t-l-bdg-get)                  ; get value from binding (of +)
    (apply)                        ; call +
  "resume26"
```

and return that. Notice that for the argument subexpressions, `compile-combo` inserts `(push)`es to save the values on the eval stack. For the first (function position) subexpression, it leaves the value in the value register, which is where it's expected (by `apply`).

The recursive call to `compile-combo` to compile `(- foo x)` goes pretty similarly to the one for `(+ bar y)`, the main difference being that both `foo` and `x` are found to be local variables and compiled appropriately, with the result being the sequence

```
    (save-continuation "resume27") ; save state for call to -
    (local-var-ref 1 2)            ; look up x
    (push)
    (local-var-ref 0 1)            ; look up foo
    (push)
    (literal-lookup 4)             ; get toplevel binding (of -)
    (t-l-bdg-get)                  ; get value from binding (of -)
    (apply)                     ; call -
  "resume27"
```

The recursive call to compile the symbol `+` goes striaghtforwardly to `compile-symbol`, which looks up `+` and finds that it's a toplevel variable; the binding is already on the literals list, so the code generated is:

```
    (literal-lookup 5)   ; get toplevel binding (of +)
    (t-l-bdg-get)        ; get value from binding (of +)
```

and this is returned to the outer invocation of `compile combo`. It can then string together the code for the outer `+` expression, putting a `save-continuation` at the front and adding an `apply` at the end. This code is returned to the inner invocation of `compile-let`, which appends it to its code and returns it to the outer invocation of `compile-let`, which returns the entire code sequence

```
    (fetch-literal 1)    ; fetch 1234
    (push)
    (fetch-literal 2)    ; fetch 3456
    (push)
    (bind 2)             ; bind 2 values (x and y)
    (fetch-literal 3)       ; get toplevel binding (of z)
    (t-l-bdg-get)           ; get value from (z's) binding
    (push)
    (bind 1)                ; bind one variable (foo)
    (save-continuation "resume26") ; save state for call to +
    (local-var-ref 1 2)            ; look up y
    (push)
    (literal-lookup 4)             ; get toplevel binding (of bar)
    (t-l-bdg-get)                  ; get value from bdg (of bar)
    (push)
    (literal-lookup 5)             ; get toplevel binding (of +)
    (t-l-bdg-get)                  ; get value from binding (of +)
    (apply)                     ; call +
  "resume26"
    (save-continuation "resume27") ; save state for call to -
    (local-var-ref 1 2)            ; look up x
    (push)
    (local-var-ref 0 1)            ; look up foo
    (push)
    (literal-lookup 4)             ; get toplevel binding (of -)
    (t-l-bdg-get)                  ; get value from binding (of -)
    (apply)                     ; call -
  "resume27"
    (literal-lookup 5)          ; get toplevel binding (of +)
    (t-l-bdg-get)               ; get value from binding (of +)
    (apply)                  ; (tail-)call +
```

## 11.10 Preserving Tail-Recursiveness using Compile-Time Continuations

One very important thing we glossed over just now when describing the workings of the compiler was when exactly to save continuations, and when not to. It is important to save continuations before calling procedures, if the callee must return and resume the execution of the caller. This is not necessary for tail calls, and in fact Scheme requires that continuations `not` be saved for tail calls—if we save continuations before tail-calls that happen to implement loops, we may end up with an unbounded accumulation of unnecessary continuations.

Another important question is when returns should be executed. If a procedure ends in a tail-call, it is assumed that the callee will do a return. But eventually something actually has to do a return, and pass control back to its caller (or the caller of its caller... whatever). This situation occurs when the tail expression of a procedure is not another procedure call, e.g., returning the value of a variable or a literal.

### 11.10.1 When Should We Save Continuations?

The general rule is that if a procedure call is the last thing a procedure does, no continuation should be saved—we can just jump into the callee, and since our state was not saved, the callee's return will resume our caller. To get this right, we just have to keep track of which expressions are being compiled in "tail position."

In the procedure

```
(lambda (x)
   (if (foo x)
       (bar (quux x))
       (baz)))
```

the `if` expression is in tail position, because the value of the if will be returned as the value of the procedure. The condition expression `(foo x)` is not in tail position, because after it is executed, control must return to this procedure so that either the consequent expression `(bar (quux x))` or the alternative expression `(baz)` can be executed.

Note that both the consequent and the alternative expressions are in tail-position; whichever is executed, that will be the last thing this procedure does, and the value computed will be the result of this procedure.

On the other hand, if we modify the procedure to always return `#f`, none of these expressions is in tail position.

```
(lambda (x)
   (if (foo x)
       (bar (quux x))
       (baz))
   #f)
```

That's because now the expression `#f` is in tail position, not the `if` expression; whatever the if does, control must come back to this procedure so that the value `#f` can be returned.

Notice that the values to compute the arguments of a combination (procedure call) are never in tail position—after computing them, control must always come back so that the procedure can be applied. The combination itself may be a tail call, of course, in which case once the arguments are computed, the apply may happen and control may never return.

To get this kind of right, all that is necessary is that each recursive call to compile should know whether the code being compiled is going to be used in tail position or not; for this we use a *compile-time continuation*. (Fear not—it's simpler than compile time environments. It's really just a flag that gets passed along to recursive calls to `compile`, sometimes getting turned off along the way.)

Keep in mind that tails of tails are in tail positions, but non-tail subexpressions are not. So in the case of nested `if`'s where the outer `if` is in tail position, only the consequent and the alternative of the consequent and the alternative are in tail position, e.g., in

```
(lambda ()
   (if (if (a)
           (b)
           (c))
       (if (d)
           (e)
           (f))
       (if (g)
           (h)
           (i))))
```

the tail calls are `(e)`, `(f)`, `(h)`, and `(i)`. All of the calls in the first inner `if` must return, because the value returned must be used by the outer `if`. The calls to the condition expressions in the other two inner ifs must also return, because the values must be used to tell which of *their* alternative and consequent to use.

For each basic kind of expression, we can tell which subexpressions should be considered tails if the overall expression is:

For a sequence, only the last subexpression can be a tail—the rest are non-tails.

For `let`, the initial value expressions for bindings are never tails, and the body is just a sequence, whose last subexpression can be a tail.

For an if, the consequent and alternative can be tails, but the condition never can.

For a procedure, the body is a sequence that's always in tail position.

When we compile something in tail position, we pass `compile` a flag saying so. The flag will be examined, and passed along to subexpressions if appropriate for compiling the kind of subexpression in question.

For example, if `compile-sequence` is handed a flag saying it should compile for tail position, it will pass the tail flag along when calling `compile` recursively on its last subexpression. For its other subexpressions, however, it will always pass the non-tail flag, because they must always return to execute the next expression in the sequence.

Similarly, `compile-if` will pass whatever flag it is given along to when calling `compile` for its consequent and alternative subexpressions, but never when compiling its condition expression.

`compile-combo` will always pass along a non-tail flag when calling compile on its subexpressions, but will examine the flag it's given to tell whether it should save a continuation before evaluating all of them.

`compile-lambda` will always compile body expressions in non-tail position, except for the last one, which is always compiled in tail position. (For simplicitly, `compile-lambda` just hands the whole body to `compile-sequence`, with a tail flag.)

`compile-let`, always compiles its initial value expressions in non-tail position, and its body expressions like a sequence. (For simplicity, it just hands the whole body to `compile-sequence`, with whatever flag it's given.)

#### 11.10.1.1 Compiling Returns

As mentioned above, when an expression other than a procedure call is a tail of a procedure, it must be accompanied by a return. That is, *every* tail of a procedure must be either an `apply` (which will call something which will return, perhaps indirectly because of tail calling) or a `return`.

The compiler can handle this by putting ensuring that wherever we generate intermediate code that is a leaf of the expression graph (e.g., in `compile-variable-ref` and `compile-literal`), we check the compile-time continuation flag to see if the expression occurs in tail position. If so, rather than simply leaving the value in the value register, we also execute a `return` sequence—a series of instructions that will grab the values out of the first partial continuation on the chain, and restore them into the registers and evaluation stack to resume the suspended procedure. We have a special intermediate code instruction that stands for this sequence, called return.

Consider the following procedure:

```
(lambda (a b c)
   (if (if a
           (b)
           c)
       d
       (e)))
```

When compiling its body, we dispatch through `compile-sequence` and recursively call `compile` to compile the `if` in tail position. It recursively calls `compile` to compile the nested if in non-tail position, which in turn recursively calls `compile` to compile `a`, `(b)` and `c` in non-tail position.

Note that `a` is a leaf expression, and since it's in non-tail position, it can just leave its value in the value register. The subsequent code (the test for false and conditional branch that's part of the code for the inner `if`) will expect that value there, so that's fine.

The expression `(b)` is not in tail position, because it inherits non-tail position from the inner `if`, so a continuation must be saved before the call to `b`. When `b` returns, its value will be in the value register and execution will resume at the branch that is part of the `if`.

Similarly, the expression `c` is in non-tail position (which it also inherited from the inner if); it can just leave its value in the value register where subsequent code can find it. (In this case, it's the value returned by the inner `if`, and tested by the outer `if`'s test for false and conditional branch.)

The expression `d` is different. It's in tail position, and it's a leaf (not a call). It can't just leave it's value in the register, because it's the end of the procedure; it must therefore have a `return` sequence tagged onto it.

The expression `(e)` is just a tail call, which can just call `e` without saving a continuation. Whatever e calls can do whatever it wants, and probably something will eventually leave something in the value register and pop the caller's continuation.

The code generated for the above procedure is:

```
    (bind 3)                        ; bind args (a, b, and c)
    (local-var-ref 0 1)               ; get value of a
    (push)
    (branch-on-false "else32")        ; compare and maybe br to inner else
    (save-continuation "resume33")
    (local-var-ref 0 2)               ; get value of b
    (apply)                           ; call b
  "resume33"
    (branch end)
  "else32"
    (local-var-ref 0 3)                   ; get value of c
  "end32"
    (branch-on-false else1)         ; compare and may br to outer else
    (fetch-literal 1)                 ; get toplevel binding of d
    (t-l-var-get)                     ; get value of d from binding
    (return)                          ; and return it
    (branch end1)
  "else31"
    (fetch-literal 2)                 ; get toplevel binding of e
    (t-l-var-get)                     ; get value of e from binding
    (apply)                           ; and tail-call it
  "end31"
```

(Notice that when we generated the code for the outer else, we generated a branch that can never be taken. `compile-if` generates a label for the end of the code, so that after executing the consequent, control will resume at whatever code follows the `if`. In the case of this `if`, the consequent will always execute a return before encountering the branch. A slightly smarter compiler would probably recognize this situation, and eliminate the branch.)

## 11.11 Compiling Top-Level Expressions

We said earlier that the compiler mainly uses recursion to generate intermediate code for nested expressions. To make this useful, though, at some point the intermediate code for a top-level expression must be converted into actual executable code and packaged up so that it can be called.

Suppose we interact with our system via a read-eval-print loop where eval is really implemented by compiling the expression and executing the resulting compiled code.

To make it easy to implement this, it's nice if there aren't very many kinds of top-level expressions that the compiler has to generate code for and be able to actually call. In particular, it's convenient if different kinds of expression can be transformed into the same kind of expression. The easy way to do this is to make all different kinds of executable expressions into expressions that generate procedures, and then call those procedures.

If we type

```
(let ((x 2))
  (+ x x))
```

to the r.e.p. loop, the r.e.p. loop can simply wrap this up in a procedure expression compile that and package it up as something executable, and call it. In effect the read-eval-print loop will convert it to

```
(lambda ()
   (let ((x 2))
      (+ x x))
```

before compiling it, and call the resulting closure to execute it.

Likewise, expressions like

```
(set! foo quux)
```

and

```
(if bar baz)
```

can be wrapped up as

```
(lambda () (set! foo quux))
```

and

```
(lambda () (if bar baz))
```

Now when we start compiling, we only have to deal with one kind of thing—a whole procedure, and when we get the resulting code back and package it up to run it, we'll always be dealing with the code for a whole procedure. That makes it easy to create an actual closure to call.

The main routine we use to start off compilation is `compile-procedure`, which takes an expression, a compile-time environment, a compile-time continuation, and a literal list as arguments. It returns intermediate code and an updated literal list for the procedure.

We take the intermediate code and hand it to the procedure `intermediate-code->executable-code` which generates the executable code object. (This may be by translating the sequence of intermediate code instructions into the equivalent sequences of assembly language instructions, and running that through an assembler. Before doing the assembly, it may run the intermediate code through one or more optimization phases.

We take the resulting executable code and the literals list, and hand them to `make-template` to create the template object.

Now we can hand the appropriate runtime environment and the template to `make-closure` and get back a callable closure for the procedure.

## 11.12 Compiling `lambda` Expressions Inside Procedures

When we compile a `lambda` expression, we must generate code that will create a closure at run time. A very naive way to do this would be to generate code that would call the compiler at runtime to compile the `lambda` expression into a procedure, plus a little code to create a closure of that object in the runtime environment.

Luckily, this is not necessary, and the compiler can do all of the real compilation at compile time—since the code for the `lambda` expression will be the same every time it's executed, and since lexical scope guarantees that it will always execute in an environment with the same structure, only one version of the code is needed, and it can be shared among all closures of that procedure. The template can be shared as well.

The compiler therefore generates code and a template for the `lambda` procedure; at run time, the actual code for the `lambda` expression just makes a closure on the heap and initializes its environment pointer and template pointer. This code will get the environment pointer from the environment register (and put it in the environment field of the new closure); the template pointer will be the ponter to the template for the `lambda` procedure.

To allow this little code sequence to quickly grab the template for the procedure being closed, the compiler stores a pointer to that template in the template of the procedure which executes the `lambda` expression. For example, if a `lambda` expression is encountered while compiling procedure `foo`, the compiler will compile the `lambda` procedure and store its template in the template of `foo`. (While compiling `foo`, it simply records the pointer to the new `lambda` procedure's template as another literal. Then it will end up in `foo`'s template like other literals.)

So the code generated for

```
...
(lambda (x)
   (...))
...
```

looks like

```
...
(envt-reg-get)     ; primitive to copy envt. reg. onto eval stack
(push)
(fetch-literal 15) ; grab template pointer for lambda proc
(push)
(make-closure)     ; code that will create closure w/those values
...
```

The real trick is in compiling the `lambda` procedure and stuffing its template into the template of the procedure that contains the `lambda` expression. The compiler just calls itself to generate the code and template then saves the template in the literal list and generates code like the above to reference the right literal.

## 11.13 Compiling Top-level Definitions

We said above that we can deal with top-level expressions by turning them all into `lambda` expressions, which will have the effect of evaluating those expressions when called.

This is a little bit tricky when dealing with top-level definitions, because top-level definitions can't be nested inside lambda expressions in the obvious way—they'd just become local definitions.

We therefore have to treat them specially. We use a special procedure, `environment-define!`, which operates on top-level environments and allows us to create top-level bindings. This is not a standard Scheme procedure—it's a special procedure that the compiler can generate calls to, but normal portable Scheme code cannot use directly.

The read-eval-print-loop will therefore recognize top-level definitions and treat them specially. When it encounters one, the initial-value expression will be wrapped up as a `lambda` and compiled, and the results turned into code, a template, and a closure. (The closure is given the runtime toplevel environment pointer.)

The closure will be called to get a result for the initial value expression, and `environment-define!` will be used to create and initialize the toplevel variable.

(This might appear at first to cause a scoping problem: if the binding isn't created until after the initial value expression is compiled, the compiler won't see the binding. But recall that if we compile an expression that uses an undefined variable, we assume it's a toplevel variable and create a binding object for it, and mark that object invalid. If the binding has already been created by a forward reference in this way, `environment-define!` will just overwrite the marker with a real value.)

Of course, if the top-level definition uses procedure definition syntax, it is necessary to massage that into a `lambda` expression before doing the above massaging and compiling.

## 11.14 Interfacing to the Runtime System

In order to support garbage collection (as is required for Scheme), there must be some agreement between the compiler and the garbage collector, so that the collector can find the pointers that the running program might find, and ensure that all objects the program could reach from them are preserved.

A common way of doing this (used in RScheme and Scheme-48) is to have a fixed set of registers (plus maybe an eval stack) that hold all of the root values that the collector needs to know about, and guarantee that whenever garbage collection occurs, all pointers will be identifiable as such. Any given register must be known to never contain pointers, to always contain a pointer, or to contain self-identifying (tagged) values that are decodable to see if they're pointers.

For example, in the straightforward compiled system we've described in detail, the `VALUE` register and the `EVAL` stack only contain normal Scheme values: tagged values that can be checked to see if they're pointers. On the other hand, the template and procedure, pointers would probably always contain raw pointers, since they can only point at one kind of thing, and the tags would slow some things down.

There might also be some other registers, which always contain nonpointers.

### 11.14.1 Garbage Collection

#### 11.14.1.1 Safe Points

Many systems (like RScheme and Scheme-48) ensure that garbage collection can only happen when a program is at a well-defined "safe point", not at an arbitrary point in the code. At a safe point, all pointer values are known to be recognizable. In between safe points, it's okay for the code to use values that aren't properly decipherable by the GC. (For example, a register that normally contains only tagged values might briefly hold a raw pointer.)

This is easy in a single-threaded system; the GC just keeps some space in reserve, so that it never runs out of memory between safe points. If an allocation requires dipping into this reserve, a flag is set so that a GC will occur at the next safe point.

The usual trick is to ensure that each procedure call and backward branch is a safe point. This ensures that the a program (or thread) reaches safe points periodically,

It's a little bit trickier in a multithreaded system—you have to make sure that you suspend threads at safe points, so that if another thread forces a GC while another thread is suspended.

#### 11.14.1.2 GC at Any Time

Some systems do not use safe points, and in effect make every point in the code a safe point for collection. They ensure that no matter where a GC occurs (or where a thread was suspended before the GC occurred), there will be enough information lying around so that the collector can find all of the pointers it needs to find.

Some compilers do this by restricting the way registers are used and code is generated. (For example, the Orbit compiler only uses certain registers to hold pointers, and only uses certain others to hold nonpointers. In addition, all pointers in registers must point directly to the beginning of an object; array indexing cannot be converted into arbitrary ponter arithmetic by the optimizing compiler.)

Other compilers allow more use of odd representations and more flexible use of registers, so that values can be figured out at run time. For example, a register might be assumed to hold nonpointers, except at points in the code flagged by the compiler, based on its having register allocated a variable there.

### 11.14.2 Interrupts

## 11.15 Advanced Compiler and Runtime System Techniques

### 11.15.1 Inlining Small Procedures

The system we've described so far generates fairly slow code, and a major culprit is the frequency of continuation saving and procedure calls. Even very small, frequently-executed procedures like eq?, car, cdr, and + require a handful if instructions to call and another handful to return, plus another handful to save a continuation if it's a non-tail call. This is much slower that the cost of similar operations in conventional languages like C or Pascal; in those languages, these simple little "operations" don't have the semantics of calls to first-class procedures.

Sometimes it is desirable to trade away some of the purity and elegance of a language like Scheme, and trade reduced flexibility for better performance. One way of doing this is by declaring frequently-used small procedures `not` to be redefinable, and allowing the compiler to compile those operations inline rather than as procedure calls. In some systems this only works for built-in procedures that the compiler understands, but in others the compiler is smart enough to inline user-defined procedures if so directed.

In some Scheme systems, you can declare procedures to be inlinable, or use a compiler flag that says you promise not to redefine the common little procedures that are most valuable to inline. This means that you can't change the definition of something like + on the fly, but you seldom want to. A common tradeoff is to avoid inlining any but the most frequently-called procedures during program development, and once the program is finished, recompile with lots of inlining. This gives you the flexibility to modify procedure definitions on the fly during debugging, while getting maximum speed once it's clear which procedures won't ever be redefined in normal operation.

Some high-tech compilers use advanced techniques to do lots of inlining when it's safe, without reducing flexibility much or requiring the user to supply a lot of declarations.

The Self compiler aggressively inlines code, and automatically recompiles the code that is invalidated by changes to procedure definitions. (This compiler is for the language Self, not Scheme, but similar techniques could be applied to Scheme.)

Some compilers currently in development have a special mode for compiling finished programs which will not be used with a read-eval-print loop. Such a compiler takes advantage of the fact that if it can look at the whole program (rather than having parts typed in by the user interactively), it can tell which variables could ever be modified at run time. (As long as there are no calls to eval at run time, the compiler can tell that all of the code for the program exists at compile-time; new closures may be created at run time, but not totally new procedures.) After globally determining that there is no code in the program that could change the definition of a procedure, it is free to inline the code for that procedure into its callers.

### 11.15.2 Type Declarations and Type Analysis

Another key performance problem with naive implementations of Scheme (or other dynamically typed languages) is that basic operations are generally slow relative to their execution in conventional statically-typed languages. For example, the Scheme procedure `+` must check the types of its arguments and (depending on those types) execute any of several possible code sequences to add two numbers. Usually, the checking overhead alone is several times greater than the cost of the actual addition.

One way of reducing this cost is by extending Scheme to allow the user to declare the types of some variables. The compiler may be able to use this information to compile fast versions of operations for values of known types. (This is especially true if common operations are inlined—the compiler can choose to inline the appropriate version rather than the more general code.)

Another way of reducing type checking cost is for the system to automatically infer the types of some expressions. For example, consider the expression `(+ a 22)`. Since `22` is a literal, its type is known at compile time. If the compiler can inline the `+` procedure, it may at least omit the type check of that argument.

A combination of declarations and inferencing can work well. For example, if the user has declared variable a to be of type `<integer>`, then the compiler can tell that `(+ a 22)` is an expression whose arguments are integers (so no run time type test are necessary there) *and* whose result is an integer, which may eliminate the need for type checks by the expression that uses the value.

More aggressive schemes are possible for reducing the frequency of dynamic type checks. For example, the Self compiler aggressively inlines and transforms code so that multiple dynamic type checks can be collapsed into a single one.

### 11.15.3 Using More Hardware Registers

[ blah blah ]

For example, it's very likely a good idea to use more registers, and either not have an eval stack or not use it as often. Our simple abstract machine requires arguments to be passed on the eval stack, which means storing into memory at least once for each argument, and loading back from memory when arguments are used. Most modern machines have several hardware registers available for argument passing, and more for holding intermediate values of computations.

If we have a few more registers that can be used for argument passing, we could just leave the argument values in those known registers, and procedures could expect them there. In many cases, argument values could be computed in a way that the result is left in the appropriate argument-passing register, without having to copy it there from somwhere else. Similarly, in many cases, procedures could leave their arguments in the argument passing registers and use them there, without actually copying them into a binding environment on the heap. (Even if only a few registers can be devoted to this, it will account for the large majority of arguments passed, since most procedure calls are to procedures that take between one and three arguments.)

Similarly, in many cases a temporary value generated by evaluating a subexpression could be left in a register, and then used by another expression, without pushing and popping the eval stack.

This can be a big performance win—it is much faster to operate on arguments and temporary values that are already in registers, rather than copying them to and from memory all of the time.

Using more registers can make the compiler and runtime system more complicated. If variables are in registers when continuations are saved, their values must be saved in the continuations and restored at procedure returns. This requires the compiler to keep track of which registers are in use at which points, and generate appropriate code. It also complicates the interface between the compiled code and the garbage collector; the garbage collector must be able to find all of the pointer values that are stored in registers, so that it can find all of the reachable objects. The compiler must therefore record sufficient information that all pointer values can be found at garbage collection time. (Alternatively, the compiler may record a safe approximation of the information, and require the collector to make conservative guesses about what's what.)

### 11.15.4 Closure Analysis

One of the performance problems with a naive implementation of Scheme is that in the general case, variable bindings must be allocated on the garbage-collected heap, and procedure calls must be via pointers to closures. This is often much slower than the usual implementation of conventional programming languages, which don't have to support `lambda`. Allocating closures and environments on the heap is mainly slow because creating and accessing variable bindings is slower than if the variables were allocated on a stack or in registers.

A smart Scheme compiler can get rid of most of this overhead by analyzing programs and noticing that many closures are used in stereotyped ways, and calls to them can be implemented more cheaply than the naive implementation. Similarly, analysis of expressions may reveal that most binding environments can't possibly be captured by closures, and therefore don't need to be allocated on the garbage-collected heap. The bindings can be saved in continuations along with temporary values, or a more conventional stack may be used, or (best of all), the bindings can be register-allocated.

A simple example of a language-level closure that doesn't need the fully general naive implementation is a closure created by a `lambda` expression that appears in the function position of a combination:

```
((lambda (x)
    (+ x x))
 2))
```

(Recall that constructs like this are often generated by macros that implement binding constructs like `let`—this one is equivalent to

```
(let ((x 2)
   (+ x x))
```

In this case, we can tell from the fact that the `lambda` expression appears in the function position that the closure can't "escape" and have anything weird done with it. That is, no pointer to the closure is assigned into a variable binding, or passed to a procedure call, or inserted into a data structure. It's clear that the only thing that can happen to this closure is that it will be called, and then the pointer to it will be "dropped," i.e., not passed anywhere else. The closure will therefore become garbage immediately after it's executed.

A smart compiler will therefore recognize that all the closure really does is bind its variable and execute it's body; it will leave out the code to create the closure and just compile in the equivalent code—in this case, it will generate the obvious code for a `let` expression.

(Some compilers always transform `let`'s and `letrec`'s into `lambda` combinations, and rely on their optimizers to recognize the unnecessary `lambda`'s and remove them. This may seem backwards, but it's nice because the same optimizations work whether the lambda combinations were the result of transforming a `let`, or macroexpanding a user-defined macro, or written directly by the user, or whatever. The more sophisticated the optimizer, the more simply the user can write macros and procedures, and expect the compiler to sort it all out and generate efficient code.)

Another simple case for closure and environment analyis is binding environments that don't have any closures created in their scopes. Suppose that our compiler inlines calls to `car`, `eq?`, and `cdr`, and consider the expression

```
(let ((x (car a))
   (if (eq? (car x) target)
       (car (cdr x))
       #f))
```

in this case, the body of the `let` can be compiled into entirely inline code, and it is clear that there is no possible path of execution that can create a closure that captures `x`. `x` can therefore be allocated in a register for its whole lifetime, making this code much faster.

[ separate section? Figure out structure here... ]

Actually, some of these analyses are trickier than they appear, due to the presence of side effects and `call/cc`.

[ Haven't talked about `call/cc` yet! ]

Consider the expression, where we don't assume any inlining

```
(let ((x (car a)))
   (if (eq? (car x) target)
       (car (cdr x))
       (set! x (foo))))
```

At first it appears that since there are no `lambda`'s in the expression, `x` can be allocated in a register, and saved in continuations across calls. (E.g., when calling `car`, we could just save the value of `x` in the continuation and have it restored when `car` returns, right?) Unfortunately, if we don't have any guarantees that `car` won't be redefined in weird ways, then it's possible that the call will be to procedure that will (directly or indirectly) call `call/cc`, and capture a continuation that could be used to return into this procedure any number of times. In that case, we can't be sure that we won't return into this code and modify `x`. If we did, then each time we returned into this environment, we should see the latest value of `x`. This will happen if the value of `x` is in a normal binding environment on the heap, but not if it's in a register that gets saved in a continuation.

Recall that when we restore a continuation, we just copy the values out into the registers. If we restore the same continuation multiple times, we'll just keep copying the same value of `x` back out.

To get this right, we have to ensure that if there are any assignments to `x`, then all references to `x` go through a pointer to a heap-allocated binding. Then when we save a continuation, we save this pointer to the binding of `x`, not the *state* (value) of the binding of `x`. Every time set or read the value of `x`, we go through this indirection to the same binding, and see the latest value.

Because of this, high-tech scheme compilers keep track of which variables are ever `set!` anywhere in their scopes, and make sure to allocate those variables' bindings on the heap.

In Scheme, it is a common idiom to code iteration as recursion; macros for different looping constructs often compile into `letrec`'s with tail-calling `lambda` expressions.

While this is a very powerful framework for expression various patterns of iteration, a naive implementation is slow. In most cases, loops created in this way are actually just used as loops, and it is desirable to compile away the overhead of closure creation and calling. For example, consider a named `let` like

```
(let loop ((x 0))
   <body>
   (if (< x 10)
       (loop (+ x 1))))
```

that has been transformed to

```
(let ((loop (lambda x)
               <body>
               (if (< x 10)
                   (loop (+ x 1))))))
   (loop 0)
```

We can look at this expression, and if no reference to the variable `loop` occurs in the `<body>` expression, we can tell that we can compile it as a loop.

The analysis here is just slightly more complicated than the one that allows us to optimize closures that are produced by `lambda` expressions in function position of a combination.

When compiling the `let`, we can keep track of each `let` variable and see whether it is ever used for as anything but the name of a procedure to tail-call—if the value of loop is never assigned, and never read except to call it, then we know that the "calls" to loop don't really need to be full-blown closure calls at all. We can inline the code for the body of the loop and compile these calls as jumps directly to that code.

FOOD FOR THOUGHT—does it matter whether the calls are tail-calls or not, if we just treat them as procedure calls to a known address, and go ahead and save a continuation with the right label?

### 11.15.5 Register Allocating Loop Variables for Loops

Notice that register closure analysis is particularly important for loop control variables and variables for little `let`'s inside loops. Because Scheme requires that a loop variable be bound again (to fresh memory) at each iteration of a loop, actually allocating them on the heap is expensive. If it can be determined that the variable is dead at the end of the loop, however, then the variable can be re-bound at each iteration by simply re-using the same register. (We're binding the name to a particular piece of memory—the register—over and over again, and it just happens that these conceptual rebindings incur no runtime cost.)

With good closure analysis, loop conversion, and register allocation, a Scheme compiler can compile "normal" loops into code that's just as efficient as any compiler's.

### 11.15.6 Conventional Optimizations

Besides the optimizations described above, conventional compiler optimizations are applicable to optimizing languages like Scheme.

Just as in a FORTRAN or C compiler, data flow analysis and control flow analysis can let the compiler simplify intermediate code and produce better machine code.

### 11.15.7 Stack Caches

Inlining and closure analyisis can greatly reduce the amount of heap allocation in a Scheme implementation. Allocating all binding environments and continuations on the heap may inflate allocation rates by an order of magnitude over the rate of allocation of normal data structures like pairs and vectors. With a simple compiler and garbage collector, this can greatly inflate garbage collection costs. Despite the high rate at which continuations and environments are allocated, there are typically relatively few of them live at any given time—the vast majority of them are used very, very briefly and then become garbage.

Inlining procedure calls may greatly reduce the allocation of continuations, and closure analysis may allow most bindings to be allocated in registers instead of on the heap.

Still, it may be desirable to keep most of the continuations and environments from making it to the normal garbage-collected heap.

A `stack cache` is an area of memory (or pool of discontiguous chunks of memory) that's used to for initial allocation of continuations and/or binding environments, in the expectation that most of them will die quickly. A stack cache caches part of the continuation chain; it's called a stack cache because it behaves mostly like a stack. Stack caches may be used for continuatons, with environments still being allocated on the heap, or a more complex design may be used to keep most environments from making it to the heap as well.

For the most part, a stack cache is treated like a stack, in that continuations are pushed and popped as though it were a stack. When a continuation is captured by `call/cc`, however, the continuation chain is first moved to the heap so that it can be preserved in the usual way. This is generally a good tradeoff, because `call/cc` is not typically executed very often, and the stack cache can behave like a stack most of the time. The large majority of continuations will be reclaimed very quickly, by popping the stack cache, while a small minority will be moved out to the normal heap.

Caching binding environments is a little trickier, but the basic principle is the same; most environments are created in the stack cache, and only moved to the garbage-collected heap when necessary, i.e., when a closure is created on the heap. At that moment, the environment is moved to the heap, one frame at a time, until a frame is reached that is already on the heap. (The code that does this must ensure that an environment is never copied to the heap twice, destroying the sharing of outer environements by inner environments created in their scope.)

It is not clear how desirable a stack cache for environments is, given a compiler that does a reasonably good job of closure analysis. Using a stack cache for environments makes closure creation slower, and if most of the short-lived environments have been eliminated by closure analysis and register allocation, it may not be worth it.

There is also some controversy about whether stack caches are worthwhile in general, or whether a generational garbage collector will take care of the large volume of short-lived data efficiently.

One interesting point is that a stack cache really *is* a kind of generational garbage collection scheme, which exploits the typically short lifetimes of particular kinds of data. (When environments and continuations are moved to the normal heap, that can be viewed as moving objects from one generation to the next. This special generation is cheaper than a normal generational scheme, however, because of the stereotyped structures of continuation chains and binding environments.)

A stack cache, because it's small, can reduce the amount of memory that is used very frequently, compared to a generational GC without a stack cache. (A stack cache may only be a few kilobytes, but the youngest generation of a generational GC may be hundreds of kilobytes, or megabytes.) For some cache architectures, frequent reuse of this large an area causes significant cache miss penalties. (For some other architectures, the misses still occur but the cost is surprisingly low. I believe that stack caches are nonetheless a good idea, because they never hurt much and may sometimes help a lot.)

Scheme-48 has a stack cache that caches both continuations and binding environments. RScheme has a stack cache for continuations only, and relies on the compiler to compile away most heap allocation of binding environments. (This may not currently be as effective as it should be—the compiler needs more testing and improvement before it will generate really good code.)

# Concept Index

# Table of Contents