Nils M Holm

Sketchy LISP

An Introduction to Functional Programming in Scheme

Third Edition

Sketchy LISP – Third Edition
Copyright (C) 2006,2007,2008 Nils M Holm
All rights reserved.
Print and Distribution: Lulu Press, Inc

ISBN 978-1-4116-7448-6

Order: www.lulu.com/content/213736

Preface

A lot has happened since the release of the previous edition of **Sketchy LISP**. The ''Six'th Revised Report on the Algorithmic Language Scheme'' (R^6RS) was ratified and Scheme is no longer the language it used to be.

This edition is dedicated to the principle that made Scheme a language that was esteemed by researchers, educators, and creative coders all around the world:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. – R^nRS introduction

Unfortunately this principle was abandoned in the R^6RS process. I created this edition in the hope that enough people will stay interested in the small and beautiful language that Scheme used to be, so the R^5RS will remain a de-facto standard

After publishing the second edition of Sketchy LISP a lot of people mailed me with really good ideas on how to improve the book. I considered all of them and implemented most of them:

This edition uses a more consistent style in the examples, which is particularly important to beginners. A style guide was added to the appendix.

A new chapter was added that illustrates what ''real-world'' Scheme code looks like by discussing a more complex program in great detail.

Some subtle flaws in the code were fixed. In particular the `case` syntax of chapter 3 now works in exactly the same way as the actual Scheme syntax.

Have fun reading the new edition!

Nils M Holm, August 2008

Acknowledgements

Thanks to all the people who pointed out subtle flaws in the code, suggested improvements, and proposed additions.

Preface to the Second Edition

This is the revised and extended second edition of **Sketchy LISP**. This edition discusses the Scheme language in much more general terms and makes fewer references to an actual implementation. The terminology is much more ''schemy'', for example ''functions'' are called ''procedures'' now and pseudo functions are explained in terms of syntax transformation.

This edition adds various topics that were not covered in the previous edition. The section about Scheme syntax has been extended and now covers macros, there is a new section about quasiquotation, and there are two additional appendices containing a table of example programs and brief summary of all Scheme functions discussed in the book.

Overall, the book has been made more general. All references to procedures that were part of a specific implementation have been removed. For example, `letrec` is explained in terms of `let` and `set!` now rather than introducing an implementation-dependent model.

Finally, the prose was revised, some typos were removed, and the text was streamlined for easier reading. I hope that you enjoy reading this new edition!

Nils M Holm, May 2007

Acknowledgements

Thanks to Diana Jeschag for catching some typos that slipped past me.

Preface to the First Edition

This book presents an overview of the Scheme programming language with strong emphasis on functional programming. Language elements and programming techniques are explained by means of simple examples which are used to form more complex programs.

Functional programming is an approach that focuses on the evaluation of expressions. Programs are formed by combining functions. Most functions are free of side effects which allows to examine programs formally and prove properties of algorithms.

The first chapter of this book introduces basic concepts such as definitions, conditional evaluation, recursion, procedures, and elementary data types.

While the second chapter continues this tour, it puts emphasis on more complex programs by introducing problems of varying complexity and then outlining the way to their solution step by step.

The last chapter takes up some lose ends and briefly introduces continuations, one of the more advanced features of Scheme. It also demonstrates how Scheme can be viewed as a formal system by constructing a Y combinator.

To make best use of this book, experimenting with the given examples is strongly recommended. See the appendix for download URLs and a brief introduction to the program development cycle.

The Scheme language achieves what only few languages have managed before: to bring fun back to programming. Its simple syntax, clean semantics, and powerful functions open the door to a fresh perspective on program design. Programming in Scheme is fun, and this book is an attempt to share some of that fun.

Nils M Holm, Feb. 2006

Acknowledgements

Thanks to Diana Jeschag for proof reading, to Al Petrofsky for explaining some of the more subtle details of binding constructs and continuations, and to Jens Axel Søgaard, Anton van Straaten, and other regulars of the `comp.lang.scheme` newsgroup for helping me to get used to the concept of continuations.

Contents

1 Basic Scheme Programming 9
1.1 Notation 9
1.2 Functional Programming 11
1.3 Loops in Functional Programs 14
1.3.1 Cond Revisited 18
1.4 Basic Data Types 21
1.4.1 Integers 21
1.4.2 Booleans 22
1.4.3 Chars, Strings, and External Representation 23
1.4.4 Lists, Symbols, and Quotation 24
1.5 Some Things You Can Do with Lists 28
1.6 Procedures and Local Scopes 31
1.6.1 What are Closures Good for? 36
1.7 Different Ways of Binding Symbols 38
1.7.1 Binding Recursive Procedures 41
1.8 Lists, Pairs, and History 44
1.8.1 Association Lists 46
1.8.2 Nested Lists 47
1.8.3 A Historical Note 51

2 Less Basic Scheme Programming 53
2.1 Variable Argument Procedures 53
2.2 Identity and Equality 57
2.2.1 A More General Form of Equality 61
2.2.2 Using Identity and Equality 66
2.3 Higher Order Functions 68
2.3.1 Some Fun with Higher Order Functions 72
2.4 Dynamic Typing 75
2.5 Type Conversion 79
2.5.1 Arithmetics with Lists 82
2.6 Arithmetics 84
2.7 Strings and Characters Recycled 88
2.7.1 Garbage Collection 96
2.8 Input, Output, and Side Effects 100
2.8.1 Input and Output 101

Contents

3 Some Missing Pieces 109
3.1 Syntax Transformation 109
3.1.1 Pattern Matching 110
3.1.2 Substitution 111
3.1.3 Recursive Syntax 114
3.2 Quasiquotation 116
3.2.1 Metaprogramming 118
3.3 Tail-Recursive Programs 120
3.4 Continuations 123
3.4.1 Non-Local Exits 126
3.4.2 Exposing some Gory Details 128
3.5 Lambda Calculus and the Y Combinator 133
3.5.1 Scheme vs Lambda Calculus 140

4 Scheme in the Wild 145
4.1 Drawing Box Diagrams 145
4.2 The DRAW-TREE Program 148

The End 155

Appendix 157
A.1 First Steps in Scheme 157
A.2 Scheme Style Guide 160
A.2.1 Definitions and Bindings 161
A.2.2 Procedure Application 164
A.2.3 Conditionals, Logic, Etc 164
A.2.4 Data and Quotation 166
A.3 Table of Example Programs 167
A.4 Scheme Syntax and Procedures 168

Index 177

1 Basic Scheme Programming

1.1 Notation

Scheme is a small language with simple syntax and clean semantics. It uses a uniform notation for all of its constructs. Here are some simple examples:

```
; Scheme notation   Math notation
(+ 5 7 9)           ; 5+7+9
(- n)               ; -n
(f x)               ; f(x)
(+ a (* b c))       ; a + b * c
```

Note that there are no precedence or associativity rules. All operations are explicitly grouped by parentheses:

```
(* (+ a b) (- c d))  ; in math: (a + b) * (c - d)
```

The semicolon introduces a *comment*. Everything between the `;` and the end of the same line is ignored by Scheme systems.

In Scheme, operators like `+` or `*` are ordinary *procedures*. A procedure is a small program that receives some values and returns a result, just like a mathematical *function*. The terms "procedure" and "function" are used as synonyms in this text.

The values passed to a procedure are called *arguments*. Because all *procedure applications* are delimited using parentheses, many procedures can have any number of arguments. For example, the expression `a+b+c+d` would be written as

```
(+ a b c d)
```

Like math functions, Scheme procedures return a result that depends on the values passed to the procedure:

```
(+ 5 7) => 12
(+ 7 9) => 16
```

The double arrow `=>` is not really part of Scheme, but it is frequently used to denote the value of an expression. In Scheme, everything that has a value is called an *expression*. `(+ 5 7)` is an expression, but `5` and `7` are also expressions. Scheme programs are formed by combining expressions.

The formula `a => b` reads "a *evaluates to* b" or "a *reduces to* b". Above formula means "*the application of + to 5 and 7 evaluates to 12*". All procedure applications have the same form: the first position between two parentheses is occupied by the procedure and the remaining positions contain arguments:

(procedure $argument_1$ ... $argument_n$) => value

Some procedures have a fixed number of arguments, some have a minimum number of arguments, and some accept any number of arguments. For instance, the " −" procedure requires at least one argument:

```
(- 5)       => -5
(- 5 7)     => -2
(- 10 2 3) => 5
(-)         => bottom
```

When a single argument is passed to it, it negates it. When more than one argument is passed to it, it subtracts all but the first of its arguments from the first one. Passing no arguments at all to "−" is an error and the application reduces to "bottom". *Bottom* denotes an undefined value. Of course, when actually entering an erroneous expression at a Scheme prompt, the system will not just print "bottom" but a more explanative message. The formula

```
e => bottom
```

merely states that something about the expression *e* is not correct. For instance, it could be used to state that the division by zero is not allowed:

```
(quotient x 0) => bottom
```

Note that above formula is not valid Scheme, because the variable *x* is not known without a corresponding declaration. It should be read as "`( quotient x 0)` reduces to bottom for any value of *x*". Such formulae are frequently used to describe the behavior of procedures in a formal way.

Most Scheme procedures are *strict*. This means that a procedure that has at least one argument of bottom also reduces to bottom:

```
(eq? (quotient 0 0) 0) => bottom
```

Although the `eq?` procedure (which compares its two arguments) *theoretically could* find out that 0/0 is not equal to 0 and reduce to falsity, it does

not. Because `eq?` is strict, it must evaluate to bottom as soon as at least one of its arguments evaluates to bottom. From a more practical point of view, this means that the evaluation of above expression would abort with an error message explaining that a division by zero was attempted. All Scheme programs abort as soon as a reduction to bottom occurs.

1.2 Functional Programming

The majority of popular languages, like C, C++, and Java, is based on the paradigm of *imperative programming*. Each program ultimately consists of statements that "do something", where "doing something" includes actions like storing values in variables, repeating sections of code, or selecting code based on conditions. Imperative languages typically include some functional aspects, too. For example, the C expression `5+7` *evaluates to* 12, just like `(+ 5 7)` does in Scheme. You can even write simple functional programs in some imperative languages like C:

```
/* Factorial function in C */
int fact(int n) {
        return n==0? 1:
                n*fact(n-1);
}
```

This function computes the factorial of its argument and returns it. There are no assignments or explicit repetitions in the program. In functional Scheme, all programs are written in this way. Each program has a number of arguments and a result. The Scheme version of the above functions looks like this:

```
(define (fact n)
  (cond ((zero? n) 1)
        (#t (* n (fact (- n 1))))))
```

`Define` looks like a procedure, but is in fact part of the Scheme syntax. Because its application looks like the application of a procedure or function, it is also called a *pseudo function*. Pseudo functions and Scheme syntax will be discussed later in detail. For now, it is sufficient to know that `define` defines a new procedure. Its general form is:

```
(define (procedure-name variable1 ... variablen)
  body)
```

The definition consists of two parts: a head and a body. The *head* of a procedure consists of the name of the procedure and the names of its *variables*. The second part is the *body* of the procedure, which is an ordinary expression.

When the new procedure is applied using, say,

```
(fact 3)
```

the variable *n* gets the value of the argument 3. In LISP terminology, you would say that *n* is *bound to* 3. [1] In this context, the procedure body is evaluated.

The `zero?` procedure is a so-called *predicate*. In LISP speak, a predicate is a procedure returning a *truth value*. `Zero?` tests whether its argument is equal to zero.

In Scheme, a predicate is a procedure returning one of the truth values `#t` and `#f`. The names of most Scheme predicates end with a question mark, like these:

```
eq?  equal?  zero?  number?
```

The outermost construct in the body of *fact* is `cond`. The `cond` pseudo function plays a central role in Scheme, becaus it controls the flow of evaluation. It is a generalized form of the `if` pseudo function which will be introduced later. A `cond` expression consists of a series of *clauses*, which are tested in sequence. Above `cond` expression has two clauses:

```
((zero? n) 1)
```

and

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

Each clause has a *predicate* and a conditional body:

```
(predicate body)
```

Both the predicate and the body are ordinary expressions. `Cond` starts by evaluating the predicate of its first clause. If the predicate evaluates to log-

1 Actually, variables are bound to storage locations rather than values, but we will use a simplified model here.

ical truth, the body of the same clause is evaluated and the result of the body is the result of the entire `cond` expression. The remaining clauses are ignored. If the predicate of the first clause reduces to logical falsity, though, `cond` proceeds with the next clause until it finds one whose predicate evaluates to truth.

It is a good idea to make the last clause of `cond` catch the remaining cases by specifying constant truth as its predicate. In Scheme, the notation `#f` represents falsity and all other values represent truth. However, it is good style to use the expression `#t` to represent logical truth. This is exactly what is done in the second clause of *fact*, so

```
(cond ((zero? n) 1)
      (#t (* n (fact (- n 1))))))
```

has the following meaning:

```
if (zero? n)
   then evaluate to 1
   else evaluate to (* n (fact (- n 1)))
```

Because `cond` is a functional construct, it returns a value: the value of the body of the first clause with a true predicate:

```
(cond (#f 0) (#t 1)) => 1
```

So what exactly happens when the above *fact* procedure is applied to a value, say, 3? Here is an answer:

```
(fact 3)
-> (cond ((zero? 3) 1) (#t (* 3 (fact (- 3 1)))))
-> (* 3 (fact (- 3 1)))
-> (* 3 (fact 2))
-> (* 3 (* 2 (fact 1)))
-> (* 3 (* 2 (* 1 (fact 0))))
-> (* 3 (* 2 (* 1 1)))
-> (* 3 (* 2 1))
-> (* 3 2)
=> 6
```

In this book, the single arrow `->` denotes ‘‘half a reduction’’. The formula `a -> b` means that *a* is *partially reduced* to *b* by reducing one or multiple sub-expressions of *a* to their normal forms. For instance, in

```
(* 2 (+ 3 4)) -> (* 2 7)
```

the sub-expression `(+ 3 4)` is reduced to its normal form, 7. The normal form then replaces the sub-expression from which it resulted, forming a new expression (which can be reduced further). Partial reduction is used to illustrate the process of reducing an expression when multiple steps are involved. At some point the reduction normally reaches a point where the expression cannot be reduced any further. Such an expression is said to be in its *normal form*. The `=>` operator *always* has a normal form on its righthand side, while `->` *never* has a normal form on its righthand side:

```
(* 2 (+ 3 4))
-> (* 2 7)
=> 14
```

The *value* of an expression is exactly equal to its normal form. Computing the value of a procedure application normally involves multiple steps.

When the body of *fact* is evaluated as shown above, each *n* in it is replaced with 3. `Cond` then evaluates `(zero? 3)` which tests whether 0=3. This is not the case, so the second clause is tested. Its predicate is `#t`, so its body is evaluated. This body contains another application of *fact*, but this time, the procedure is applied to `(- 3 1)` = 2. The same path is taken over and over again until *n* reaches zero. Finally, `(zero? n)` returns truth and so `fact` evaluates to 1. The resulting 1 replaces the application of *fact* in the expression

```
(* 3 (* 2 (* 1 (fact 0))))
```

As soon as `(fact 0)` has returned its value, the complete expression can be reduced to 6 by performing the multiplications represented by `*`.

The sample reduction of *fact* above shows one major advantage of functional languages over imperative languages. Programs can be ‘‘executed’’ on a sheet of paper by re-writing them according to a set of simple rules. This property makes it easy to verify characteristics of programs formally. Try this with a C program.

1.3 Loops in Functional Programs

Typical Scheme procedures have much in common with math functions. Their sole purpose is to map a number of values (its arguments) to another value, the function result. For example the math function

$f(x) := x^2$

would look like this in Scheme:

```
(define (f x) (* x x))
```

But of course you would give it a more descriptive name, like *square*. Even the *fact* procedure given in the previous section is modelled after a math function:

$0! := 1$
$n! := n \times (n-1)!$

But instead of declaring multiple instances of the function that handle different argument values, like in the math definition above, `cond` decides which path to take:

```
(define (fact n)
  (cond
    ; (fact 0) := 1
    ((zero? n) 1)
    ; (fact n) := (* n (fact (- n 1)))
    (#t (* n (fact (- n 1))))))
```

You probably have noticed that the definition of *fact* is self-referential. It applies itself in order to compute its own result. This is by no means contradictory. It is a well-known principle known as *recursion*. A recursive procedure has (at least) two cases: the trivial case and the general case (also called the recursive case). Above definition could be pronounced as

"The factorial of N is
– 1, if N equals 0
– N times the factorial of N-1, otherwise."

The condition "*if N equals 0*" is very specific, while the condition "*otherwise*" is rather general. Also, the first case returns the value 1 which is rather trivial, while the value of the second case is computed recursively and hence not *that* trivial.

As it happens, the trivial case of a recursive procedure is exactly what you would specify as an exit condition in a loop of an imperative program, as the following C program illustrates:

```
/* Imperative fact() in C */
int ifact(int n) {
        int r = 1;
        while (!(n == 0)) {
                r = r * n;
                n = n-1;
        }
        return r;
}
```

The loop in this program reads: "*while the exit condition (n==0) is* **not** *true, repeat the general case.*" However, this program does not tell you much about the nature of the *fact* procedure. All it does is to shuffle some state until an exit condition is met and then return some of that state. Reducing `ifact(3)` on a sheet of paper would be a tedious exercise.

What is more interesting, though, is that iteration and recursion seem to be equivalent to some degree. In fact, this equivalence is strong enough that *Scheme expresses any repetition using recursion.* This approach has some advantages:

- It allows for a declarative style of programming;
- It frees you from keeping track of a lot of state;
- It facilitates the analysis of algorithms.

An issue *seems to be*, though, that recursion looks inefficient in some situations. If you are used to imperative languages, you may have heard that recursion is bad because it slows down programs and may even break things (for example by overflowing the runtime stack). *Relax. In Scheme this is not the case.*

Both the recursive C program and the recursive Scheme program call *fact* for the same number of times. (To "call a procedure" is a more technical term for applying a function.) Each call causes the space consumed by the program to grow by some constant amount. The total amount is this constant times the size of the argument: `(fact 5)` adds five levels of recursion and `(fact 100)` adds one hundred levels. The `ifact` function, on the other hand, does not add anything. It does so by cheating. It adds some state, namely the variable *r*, to hold the intermediate result of the computation. Scheme would be a boring language, if it would not support the same sort of cheat:

```
(define (fact2 n r)
  (cond ((zero? n) r)
        (#t (fact2 (- n 1) (* n r)))))
```

The *fact2* procedure still recurses, but its space remains constant while it evaluates:

```
(fact2 5 1)
-> (fact2 4 5)
-> (fact2 3 20)
-> (fact2 2 60)
-> (fact2 1 120)
-> (fact2 0 120)
=> 120
```

What looks fine on a sheet of paper also works on a real CPU. The principle that makes this possible is called *tail call optimization.* In *fact2* the body of the general case is

```
(fact2 (- n 1) (* n r))
```

When *fact2* returns from the recursive call, the only thing that is left to do for the calling procedure is to pass control back to its own caller. Scheme recognizes this and transforms the call to *fact2* into a jump. The flow of control never returns to the caller (which would be *fact2* itself), but directly to the caller of the caller.

A program that ‘‘does not do anything’’ after calling itself is called a *tail recursive* program. *Fact2* is a tail-recursive program, while the original *fact* program is not, because it applies `*` to the value returned by the recursive call:

```
(* n (fact (- n 1)))
; ^^^--- this is done after the return of fact
```

Tail call optimization transforms tail-recursive programs into programs doing ordinary loops. It combines the expressiveness of recursion with the advantages of iteration. If you do not believe this, try the following small program which implements an indefinite loop by omitting the trivial case:

```
(define (o) (o))
```

Once started `(o)` will recurse forever, but its space will remain constant.

This is why tail-recursive Scheme programs are said to "*evaluate in constant space*".

All linear-recursive programs (like *fact*) can be transformed into tail-recursive programs using an additional argument. Note that this transformation does *not* turn the program into a total mess like the transformation of the C program. The Scheme program still is declarative:

```
(fact2 0 r) := r
(fact2 n r) := (fact2 (- n 1) (* r n))
```

The only thing that is a bit ugly is that *fact2* takes two arguments, where the second one always must be 1. This fact can be easily covered, though, by wrapping a new definition of *fact* around *fact2*:

```
(define (fact n) (fact2 n 1))
```

If you do not like to write two procedures where only one is required, `letrec` is your friend:

```
(define (fact n)
  (letrec
    ((fact2
       (lambda (n r)
         (cond ((zero? n) r)
               (#t (fact2 (- n 1) (* n r)))))))
    (fact2 n 1)))
```

`Letrec` embeds local definitions into a procedure. In the above example, it defines *fact2* locally inside of *fact*. Do not worry too much about `letrec` and `lambda` for now. They will be explained soon.

By the way: while `ifact(100)` is a rather hypothetical construct in C and will deliver anything but the result of 100!, this works fine in Scheme:

```
(fact 100)
=> 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

1.3.1 Cond Revisited

While `cond` may give the impression that it is used to write programs in a declarative way, there is one important difference: *the order of clauses*

does matter, while the order of declarations does not. For example, the declarations

0! := 1
n! := n×(n-1)!

are equivalent to

n! := n×(n-1)!
0! := 1

because you would pick the best match before re-writing an application of the factorial function. Even if the case *n!* also covers *0!*, you would probably prefer the more special case, and this would be the right thing to do. `Cond` is not that smart. It simply tests its clauses in sequence and picks the first one with a true predicate. Therefore the following program will **not** compute *n!*:

```
; A broken version of fact
(define (broken-fact n)
  (cond (#t (* n (broken-fact (- n 1))))
        ; The following part is unreachable!
        ((zero? n) 1)))
```

A cure for this problem is to always specify clauses of `cond` in order of ascending generality. The most general clause, usually containing a `#t` predicate, *must always be the last clause*. Remember: clauses after a true predicate are unreachable. Ordering the clauses of more complex applications of `cond` requires some practice, but not more than formulating complex exit conditions in imperative languages.

The keyword `else`, which is part of the `cond` syntax, helps to remember that the catch-all clause should be last:

```
(define (fact n)
  (cond ((zero? n) 1)
        (else (* n (fact (- n 1))))))
```

Whenever `else` appears in the last clause of `cond`, it is equivalent to `#t`. Its value is undefined in other clauses:

```
(cond (else 1)
      (else 2)) => bottom
```

Another interesting fact about `cond` is that it is non-strict. If it was strict, it could not work. Remember the simple procedure *o* which never terminates:

```
(define (o) (o))
```

Once applied, *o* keeps evaluating **for ever** (or until your computer breaks or until you get bored and abort program execution, whatever comes first). Because there is no way to obtain the value of `(o)`, it is *undefined*, which is the same as *bottom*:

```
(o) => bottom
```

The following expression evaluates to a value, though:

```
(cond (#f (o)) (#t 1)) => 1
```

Earlier in this book, the equivalence between `cond` and `if` was illustrated. Given this equivalence, the above expression could be re-written as

```
if #f
   then evaluate to (o)
   else evaluate to 1
```

Scheme actually does have an `if` function, which looks like this:

```
(if predicate consequent alternative)
```

It is just another way of saying:

```
(cond (predicate consequent) (#t alternative))
```

If `if` was a strict function,

```
(if #f (o) 1) => bottom
```

because

```
(o) => bottom
```

So `if` cannot be strict. The syntax of `cond` hides the fact that `(o)` is an argument of `cond`, but the equivalence

```
(cond (p e1) (#t e2))  =  (if p e1 e2)
```

demonstrates that `cond` cannot be strict, either.

The practical meaning of this property of `cond` (and `if`) is that it is a pre-

requisite for recursive functions. The procedure *down*, which counts down to 0, could not be implemented using a hypothetical strict `if` function:

```
(define (down x)
  (if (zero? x)
      0
      (down (- x 1))))
```

Of course, `if` does not really evaluate the general case to *bottom* and then ignores it, but it evaluates the general case if, and *only* if, the predicate did not hold.

1.4 Basic Data Types

Up to this point only two data types have been introduced: the integer and the boolean. There are more types in Scheme, though. Some of them will be introduced in this section.

1.4.1 Integers

The *integer* is the only numeric type that is required to be supported by all Scheme implementations. Most implementations also provide floating point, rational and complex numbers, but these are not discussed in detail here. While Scheme uses a highly interesting and mathematically correct model for numeric computations involving floating point arithmetics, most of the numeric code in this book deals with integers.

One interesting property of Scheme arithmetics is that computations do not overflow. There are still some popular languages that will silently provide a wrong result for expressions like

```
1234567890 + 9876543210
```

For instance, the following C program will print -1773790788 (on a two's-complement 32-bit machine), which is plain wrong:

```
#include <stdio.h>
int main(void) {
        printf("%ld\n", 1234567890 + 9876543210);
        return 0;
}
```

Of course, modern computers have 64-bit integers, but there will always

be a number that is too big to compute in a given number of bits, and there will always be a language which computes the wrong result without notice.

Scheme uses so-called *bignum arithmetics*. "Bignum" is short for "big number". The implementation of this kind of arithmetics is independent from a specific kind of hardware. It can handle all numbers that fit in memory, so the only sort of overflow that theoretically can occur while doing bignum arithmetics is the memory overflow, and even such an event will reliably abort the computation in progress and not deliver a wrong result.

Some versions of Scheme use bignum arithmetics exclusively, but more efficient implementations use native arithmetic operations as long as the result fits in a machine word. When an overflow occurs, they switch to bignum mode. To the user there is no visible difference. Bignum integers look exactly the same as "native" integers. As long as the result of an operation fits in a machine word, Scheme is about as fast as any other language. When an overflow occurs, it does slow down, but still produces correct results.

1.4.2 Booleans

The *boolean* is used to represent logical truth and falsity. There are only two literals representing theses states, which is why this sub-section is rather short.

The object `#t` represents logical truth (or just "truth") and `#f` represents logical falsity ("falsity", the "false" value). There is only one false value, but any object except for `#f` is considered a true value. This is why the following expression works fine:

```
(cond (0 'foo)) => foo
```

Do not worry about the sub-expression `'foo` – it will be explained real soon now. What is more interesting is the fact that 0 is considered to be true. This might look odd at a first glance, but it simplifies some code significantly.

For example, the `member` procedure searches a given object in a list. When it finds a matching member, it returns a sublist with that member at the first position. When it does not find a match, it returns `#f`. Because all non-`#f` values are considered true, you can write code like this:

```
(define (digit-type x)
  (cond ((member x '(1 3 5 7 9)) 'odd)
        ((member x '(0 2 4 6 8)) 'even)
        (else                    'not-a-number)))
```

1.4.3 Chars, Strings, and External Representation

Like most other languages, Scheme provides data types for holding single characters and sequences of characters. Single characters are represented by *chars*, and sequences of characters are represented by strings.

At this point, it is time to introduce something called *external representation*. The external representation of an object is what the programmer keys in in order to create an object of a given type. For instance, 123 is typed to create a numeric object with a value of 123. Of course, the computer does not really store the number 123 in this form, but it converts it to some *internal representation* that is more efficient from the machine's point of view.

Because the internal representation of 123 would be something like 01111010, which is hard to decode for a human being, each Scheme object maps to exactly one single external representation. Numbers map to their decimal forms, logical falsity maps to `#f`, etc.

The Scheme procedures `read` and `write` translate between internal and external representation. `Read` maps external representation to the internal one:

```
(read) 12345 => 12345
```

Because Scheme environments use `write` to output their results, though, you cannot see the internal representation. In fact, the internal representation is *never* of any interest to Scheme programmers. You do not have to care about things like byte ordering, memory layout, etc. Scheme hides these details from you and allows you to concentrate on real problems instead.

The external representation of a char consists of the prefix `#\` and the character to represent, for example:

```
#\H #\e #\l #\l #\o
#\\ #\( #\5 #\+ #\
```

The last char in the above sample represents the blank character, but this representation is unfortunate, because the character to represent is invisible. This is why an alternative external representation of this character exists: `#\space`. The same is valid for the newline character, which is written `#\newline`.

A sequence of characters is called a *string*. The characters of a string are enclosed in double quotes:

```
"Hello, World!"
```

To include a quotation mark in a string, it must be prefixed with a backslash:

```
"He said \"hello\"."
```

To include a backslash, it must be prefixed with another backslash. The external representation of a string requires the backslash before the characters `#\"` and `#\\`. Therefore,

```
"\"Hi\"" => "\"Hi\""
```

If you want to output a string without the additional backslashes, you have to use the pretty-printing `display` procedure:

```
(display "A\\B")  writes  A\B
```

Note that the output of `display` is *not* an external representation. When discussing Scheme, the external representation is used exclusively to refer to objects. Neither the sequence `A\B` nor the sequence `"A\B"` is a valid object of the type string.

1.4.4 Lists, Symbols, and Quotation

In fact I was lying when I told you that you only had learned about two data types at the beginning of this section. Actually, it were four, because programs like

```
(define (fact n)
  (cond ((zero? n) 1)
        (else (* n (fact (- n 1))))))
```

are nothing but *lists* of integers, booleans, other lists, and *symbols*. So you already have seen quite a few lists and symbols in this book. You even

may have understood how lists work by using your intuition. A simple explanation would be that a list is a sequence of objects enclosed by parentheses, like these:

```
(1 2 3)
(#t "hi" (+ 5 7) #\x)
(cond (a b) (#t #f))
(define (id x) x)
() ; <--- this is the empty list
```

These examples ilustrate two notable properties: The first one is that a list may contain lists. Lists may be nested to any depth. The second one is that Scheme programs *are* in fact lists. This is a particularly nice feature, because it allows Scheme programs to examine, decompose, and even *write* Scheme programs in a simple way, just by using procedures for manipulating lists. On the other hand, this may seem troublesome, because procedure applications look like lists, too:

```
(+ 5 7)
```

Is this supposed to be a list of three elements or an application of the `+` procedure to the numbers 5 and 7? At this point, *quotation* comes into play. Above expression is not quoted, so it is a procedure application:

```
(+ 5 7) => 12
```

To create a list containing the members `+`, 5, and 7, the `quote` pseudo function is used:

```
(quote (+ 5 7)) => (+ 5 7)
```

`Quote` is a *very* important and frequently used construct and albeit it appears to do nothing more than return its argument without any further evaluation, you should read about its implications carefully. *Understanding* `quote` *is absolutely essential in order to understand Scheme.*

Another type that frequently has to be quoted is the symbol. While symbols are common in most programming languages, quoted symbols are only present in a minority of them. As long as a symbol occurs without quotation in a Scheme program, it has exactly the same meaning as in most other languages: either it is a variable representing a value or it is a keyword.

Scheme has keywords, too. You already have seen a few in this book, namely `define`, `cond`, `if`, `letrec`, and `lambda`. Most other symbols you have seen were in fact variables, including, for instance, `zero?`, `fact`, and `+`.

As you can see, the naming conventions for symbols are rather lax in Scheme. You may include letters, numbers, and even a wide range of special characters, like `+ - * / : < > ?` and others.

Characters you may *never* use in symbols are these:

```
( ) [ ] . ' ` , # " ;
```

You may not use sequences that could be mixed up with numbers, either (like 123, -1, etc).

The `define` pseudo function introduces a variable, *binds* it to a storage location, and makes that location refer to the normal form of an expression. In most contexts, it is also correct to say that it ''binds a variable to a value''.

Note that variables do not have types in Scheme, so this is different from storing a value in a typed variable. When a value is stored in a typed variable, like in a C program, the variable must have the right type to hold that value. You cannot store a C string in a C int. A Scheme variable, however, is merely a reference to a value. By associating a variable with a value, you make the variable refer to that value.

Once a variable is associated with with a value, each reference to the variable evaluates to that value:

```
(define foo 5)
foo => 5
(+ foo foo) => 10
(+ foo (* foo foo)) => 30
```

It is an error to refer to a symbol that is not bound to any value:

```
unbound-symbol => bottom
```

Unlike most other popular languages, Scheme can refer to symbols *themselves*. In order to refer to a symbol rather than its value, the symbol is quoted:

```
        foo => 5
(quote foo) => foo
```

Even unbound symbols can be referred to in this way:

```
(quote unbound-symbol) => unbound-symbol
```

To avoid misunderstandings, the term *symbol* will be used to denote quoted symbols and unbound symbols in this book. Unquoted and bound symbols will be referred to as variables or arguments or, if they are bound to procedures, even as procedures, as in "*the procedure f takes two arguments a and b*".

An alternative notation for quoting expressions is:

```
'foo => foo
```

This is because typing `(quote something)` each time you want to refer to a symbol rather than its value or create a list rather than apply a procedure is a bit tedious. The two notations are perfectly equivalent:

```
'expression = (quote expression)
```

(Quoted) symbols do not have any value other than their name. They cannot be composed, decomposed, or manipulated in any other way. All you can do with them is to convert them to strings:

```
(symbol->string 'foo) => "foo"
```

or compare them:

```
(eq? 'foo 'foo) => #t
(eq? 'foo 'bar) => #f
(define baz 'foo)
(eq? 'foo baz) => #t
```

The `eq?` predicate tests whether its two arguments are *identical*. The term "identical" is much stronger than the terms "equal" or "equivalent". It states that two appearances of an object are in fact *one and the same.* All occurrences of one symbol are *identical*. Applying `eq?` to two appearances of the same symbol always yields truth. Symbols, the empty list `()`, and the truth literals `#t` and `#f` are the only objects whose occurrences are identical.

Although there are many Scheme programs that do not make any use of

symbols, they are essential to the language, because Scheme programs themselves are made of lists and symbols. For example, the expression

```
'(define (fact n)
   (cond ((zero? n) 1)
         (else (* n (fact (- n 1))))))
```

reduces to a list that resembles a Scheme program. Note that the quote character (`'`) at the beginning quotes the *entire* expression. The expression consists of (lists containing) truth literals, integers, and lots of symbols. You can use list procedures to decompose the program and examine its components. For example, you can extract the first member of the list, `define`, and store it in a variable *x*. You can then use the expression

```
(eq? x 'define)
```

to find out whether the program defines something. This is how Scheme programs can examine Scheme programs. The necessary procedures for composing and decomposing lists will be introduced in the following section.

1.5 Some Things You Can Do with Lists

Scheme is a language of the LISP family, and "*LISP*" is short for "LISt Processor", so it is not surprising that Scheme has quite a few procedures for processing lists. The list is a simple yet very flexible data structure that can be used to implement a lot of useful algorithms. In the previous section, the list was described as a sequence of objects delimited by parentheses. While this is true, it is a simplified description. This section will explain lists a little more in depth.

The most basic procedures for processing lists are called `cons`, `car`, and `cdr`. `Car` and `cdr` are used to decompose lists and `cons` is used to *cons*truct lists:

```
  (car '(a b c)) => a
  (cdr '(a b c)) => (b c)
(cons 'a '(b c)) => (a b c)
```

The following relation exists between these procedures: For any non-empty list *x*, the subsequent proposition holds:

```
x  =  (cons (car x) (cdr x))
```

The `car` procedure extracts the first member of a list. This member is also called the *head* of the list. `Cdr` extracts a sublist containing *all but the head* of a list. This sublist is called the *tail* of the list. `Cons`, finally, assembles a new list by attaching a new head to an existing list.

There are a few important things to notice about these procedures. For example, `car` always extracts the first member of a *flat* list, where lists contained in lists count as single members, so:

```
(car '((a b) c)) => (a b)
```

The same is valid for `cdr`:

```
(cdr '((a b) d)) => (d)
```

`Cons` can only add one single member to the head of an existing list. It cannot be used to append lists, because

```
(cons '(a b) '(c d)) => ((a b) c d)
```

If you want to append lists, use `append`:

```
(append '(a b) '(c d)) => (a b c d)
```

The tail of a single-element list is the *empty list* and adding an element to the empty list results in a single-element list:

```
(cdr '(x)) => ()
(cons 'x '()) => (x)
```

Neither `car` nor `cdr` may be applied to the empty list, because the empty list has neither a head nor a tail:

```
(car '()) => bottom
(cdr '()) => bottom
```

Using just the three basic list procedures, you can do a lot of interesting things, already. For instance, an append procedure can easily be written in terms of these:

```
(define (append2 a b)
  (cond ((null? a) b)
        (else (cons (car a)
                    (append2 (cdr a) b)))))
```

The `null?` predicate tests whether a list is empty:

```
(null? '(a b)) => #f
(null? '())    => #t
```

(Note that the empty list *must* be quoted in standard Scheme, even if some implementations do not enforce this rule.)

So how does *append2* work? Here is an explanation:

```
(append2 '(a b) '(c d))
-> (append2 (a b) (c d))
-> (cons (car (a b)) (append2 (cdr (a b)) (c d)))
-> (cons a (append2 (b) (c d)))
-> (cons a (cons b (append2 () (c d))))
-> (cons a (cons b (c d)))
-> (cons a (b c d))
=> (a b c d)
```

Note that Scheme notation does not provide any means of representing intermediate steps involving symbols or literal lists. Therefore, literal objects are outlined using boldface characters in partial reductions. For example,

```
(cons a (b c d))
```

should actually be read as

```
(cons 'a '(b c d))
```

in illustrations like the sample reduction of *append2* above.

You may have noticed that *append2* has the same weakness as the *fact* procedure introduced at the beginning of this book: it adds one recursive call for each element in the first list. Can this procedure, too, be converted into a tail-recursive one? Yes, it can:

```
(define (append2 a b)
  (letrec
    ((app2
       (lambda (a b)
         (cond ((null? a) b)
               (else (app2 (cdr a)
                           (cons (car a) b)))))))
    (app2 (reverse a) b)))
```

The embedded *app2* procedure, which does the real work, removes one member of *a* at each iteration and attaches it to *b*. However, this causes the

members of *a* to be in reverse order when the procedure finishes. This is why the list *a* is *reversed* before passing it to *app2*. This is quite a common technique in Scheme.

This version of *append2* is much better than the initial one, but still not as nice as the `append` procedure that is built into the language. For example, to append three lists, you have to write

```
(append2 '(a b) (append2 '(c d) '(e f)))
```

while the built-in procedure lets you append any number of lists:

```
(append) => ()
(append '(a b)) => (a b)
(append '(a b) '(c d)) => (a b c d)
(append '(a b) '(c d) '(e f)) => (a b c d e f)
```

And this version of `append` can still be written in Scheme! How this is done will be explained in a later section, though.

1.6 Procedures and Local Scopes

You may have wondered what the `lambda` in procedures like *fact2* is about:

```
(define (fact n)
  (letrec
    ((fact2
       (lambda (n r)
         (cond ((zero? n) r)
               (else (fact2 (- n 1) (* n r)))))))
    (fact2 n 1)))
```

Again, there is a simple explanation and a more complex one. The simple explanation is that ‘‘lambda’’ is just a placeholder name for anonymous procedures. For instance,

```
(lambda (x) (* x x))
```

creates an anonymous procedure that computes the square of its argument:

```
(lambda (x) (* x x)) => #<procedure (x)>
```

The question what exactly a procedure is is part of the more complex answer and will be discussed a bit later. What is more important now is the fact that there is not much of a difference between named procedures and anonymous procedures. You can apply lambda functions to arguments just like any ordinary procedure:

```
((lambda (x) (* x x)) 7) => 49
```

You can even bind an anonymous procedure to a symbol, making the anonymous procedure a named procedure:

```
(define square (lambda (x) (* x x)))
```

In fact the above definition is *perfectly equivalent* to

```
(define (square x) (* x x))
```

and in early versions of Scheme the variant involving `lambda` was the only way to define new procedures. `Letrec` does not support the new syntax, so binding an anonymous procedure to a name is the only way to create a local procedure inside of it. So much for the simple answer.

All Scheme procedures are either *primitive procedures* or *closures*. Primitive procedures are procedures that can not (or, at least, not easily) be implemented in Scheme. They form the very core of the language, and they are mostly implemented in some other, typically more low-level, language. The following procedures are examples for ''primitives'':

```
car => #<primitive car>
cons => #<primitive cons>
write => #<primitive write>
```

Constructs like `lambda`, `define`, and `cond` are also primitives in the sense that they are part of the core language, but they are not really procedures at all. They are part of the syntax of the language.

Some people may tell you that `lambda` is the only primitive that is really needed to build a working Scheme. While this is true and even may be a challenging mental exercise, it is not very practical, so do not listen to these people until you feel a bit more familiar with Scheme.

Whether a pre-defined procedure is a primitive procedure or a closure is merely an implementation detail. Some Scheme systems do not even distinguish between them:

```
reverse => #<procedure (x)>
(lambda (x) x) => #<procedure (x)>
```

BTW: All external representations that begin with the prefix `#<` are *unreadable*, which means that they cannot be parsed by the `read` procedure. This is because there is no practical external representation for procedures. What should `write` output for a primitive procedure? Its machine code? That would not be very portable. The same is valid for closures. Their implementation depends highly on the Scheme system in use, so they do not have a universal external representation, either.

What exactly is a closure? To answer this question in detail, another pseudo function will be explained first: `let`. This construct is a close relative of `letrec`, which has been used to create local procedures in some example programs so far. `Let` binds symbols locally, creating a new context. The following Scheme session illustrates how it works:

```
(define a 'foo)
(define b 'bar)
(let ((a 3)
      (b 4))
  (* a b))
=> 12
a => foo
b => bar
```

First *a* is bound to *foo* and *b* to *bar*. `Let` then binds *a* to 3 and *b* to 4 locally, thereby creating a new context. In this context, `*` is applied to *a* and *b*. When the `let` finishes, the original bindings of the symbols that were re-bound locally are restored. The context outside of a `let` is also called the *outer context* of that `let` and the context created by it is called its *inner context*. Variables of the inner and outer context can be merged:

```
(define foo 100)
(let ((bar 70))
  (- foo bar))
=> 30
```

Only when an inner and an outer variable have the same name, the outer binding becomes invisible inside of an inner context. Symbols that are unbound in an outer context remain unbound after their use in an inner context:

```
xyz => bottom
(let ((xyz 'some-value))
  xyz)
=> some-value
xyz => bottom
```

A variable that occurs in an argument list of a lambda function or in the definition part of `let` (or `letrec`) is said to be *bound* in that procedure or binding construct. For instance, the symbol *x* is bound in

```
(lambda (x) x)
```

and in

```
(let ((x 'foo)) x)
```

but it is not bound in

```
(lambda (y) x)
```

A variable that is not bound in a given context is said to be *free* in that context. Note that a variable can be bound in one context and free in another at the same time:

```
(lambda (x) (lambda (y) (cons x y)))
```

In this example, *x* is bound in the whole expression, but *y* is only bound in the inner lambda function. *X* is free in the inner function alone:

```
(lambda (y) (cons x y))
```

A variable is called ‘‘bound’’ in a given context, like a lambda function, because it is bound to a value when the procedure is applied. Free variables, on the other hand, remain unchanged when the procedure is called. The values of free variables are defined in the outer context of the procedure.

`Let` is very similar to a procedure application: it binds values to local symbols (which are therefore ‘‘bound in its context’’), evaluates a body, restores the original bindings of its local variables, and returns the normal form of the body. This is exactly what happens during a procedure application:

```
((lambda (a b)    (let ((a 5)
   (* a b))             (b 7))
 5 7)                 (* a b))
=> 35             => 35
```

So `let` is just an alternative syntax for the application of lambda functions:

```
((lambda (a1 ... an)  =  (let ((a1 v1)
   body)                          ...
 v1 ... vn)                       (an vn))
                                body)
```

One *major* difference between `let` and the application of a lambda function is that the application can be decomposed into two parts: the lambda function itself and the list of its arguments:

```
(car '((lambda (a b) (* a b)) 3 4))
=> (lambda (a b) (* a b))
(cdr '((lambda (a b) (* a b)) 3 4))
=> (3 4)
```

An interesting question to ask at this point would be: what is the outer context of the lambda function? The inner context is created when the procedure is applied to its arguments, but this application may take place a long time after defining the procedure and the outer context might have changed in the meantime. For example, what is the following program supposed to evaluate to:

```
(let ((v 1))
  (let ((f (lambda (x) (* x v))))
    (let ((v 0))
      (f 5))))
```

In the outermost `let`, *v* is assigned to one. In this context the procedure *f* is created. In the context of *f*, *v* is bound to zero. In this context, finally, *f* is applied to five. To find out what `(f 5)` reduces to, the value of *v* in *f* must be known. The fact that the innermost binding always takes precedence suggests that the expression evaluates to zero, but in fact it reduces to five:

```
(let ((v 1))
  (let ((f (lambda (x) (* x v))))
    (let ((v 0))
      (f 5))))
=> 5
```

How can this be? Because the outer context may change between the definition of a procedure and its application, `lambda` "freezes" the outer

context at the time of its application and takes it with it. This is what the experienced LISPer calls *lexical scoping*. It is called ''lexical'' because the value of a variable depends on its lexical (or textual) context. *F* is created in a context where *v* is bound to one, so appearances of *v* in the body of *f* evaluate to one.

BTW: If the above expression would have reduced to zero, the ''dynamic'' value of *v* would have been used. This principle is known as *dynamic scoping*, but Scheme uses lexical scoping exclusively.

The ''frozen'' context of a procedure is also called the *lexical environment* of that procedure. The combination of a lambda function and a lexical environment is called a closure. In Scheme, every (non-primitive) procedure is a closure, so the terms closure and procedure are mostly used as synonyms.

1.6.1 What are Closures Good for?

In case you wonder what the use of lambda is: it can be used to create new procedures on the fly. Here is a rather classic example:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

The *compose* procedure takes two arguments *f* and *g* (which must be procedures themselves). It evaluates to a procedure of one argument that applies the two procedures passed to *compose* to its own argument (*x*). The variables *f* and *g* belong to the outer context of the `lambda`, so they are captured by the resulting closure. One says that `lambda` *closes over f* and *g*:

```
(compose car cdr) => #<procedure (x)>
                     ; body = (f (g x))
                     ; f = car, g = cdr
```

The resulting procedure applies the composition of `car` and `cdr` to its argument, thereby extracting the second member of a list:

```
((compose car cdr) '(a b c)) => b
```

BTW: The procedure `(compose car cdr)` is part of the Scheme standard. Its name is `cadr`. Of course, you could define `cadr` in a much more straight forward way:

```
(define (cadr x) (car (cdr x)))
```

So the above example is indeed a bit artifical. Here is another example: Imagine a procedure named *filter* which extracts all members with a given property from a list:

```
(define (filter p a)
  (cond ((null? a) '())
        ((p (car a))
          (cons (car a)
                (filter p (cdr a))))
        (else (filter p (cdr a)))))
```

Filter uses a predicate to check whether a member has a given property. All members *x* which satisfy `(p x)` are collected in the resulting list. The following sample application of *filter* extracts all numbers from a list (the `number?` predicate tests whether its argument is a number):

```
(filter number? '(a 31 b 41 c 59)) => (31 41 59)
```

If you want to extract all non-numbers from the same list, you could pass a lambda function negating the predicate `number?` to *filter*:

```
(filter (lambda (x)
          (not (number? x)))
        '(a 31 b 41 c 59))
=> (a b c)
```

However, the above lambda function leads to another interesting procedure:

```
(define (complement p)
  (lambda (x) (not (p x))))
```

The *complement* procedure creates the complement *P'* of a predicate *P*. Wherever *P* returns `#t`, *P'* returns `#f` and vice versa. Using *complement*, above application of *filter* can be written like this:

```
(filter (complement number?) '(a 31 b 41 c 59))
=> (a b c)
```

Complement can also be used to create a procedure that collects all members of a list which do not satisfy the given predicate, thereby effectively removing all matching members:

```
(define (remove p a)
  (filter (complement p) a))
```

which would allow you to write:

```
(remove number? '(a 31 b 41 c 59)) => (a b c)
```

1.7 Different Ways of Binding Symbols

In the previous section, two facilities for binding symbols locally were mentioned, but there is a third one, so Scheme has in fact three variants of `let`:

```
let let* letrec
```

This section explains the differences between these constructs. `Let` is the most straight-forward one of them. As outlined before, it is basically equal to a procedure application:

```
(let ((x 3)) (* x x))
```

is just another way of writing:

```
((lambda (x) (* x x)) 3)
```

Procedure applications are typically divided into the procedure part (which is bound to a name) and the arguments to which the procedure is applied:

```
(define square
  (lambda (x) (* x x)))
(square 3) => 9
(square 5) => 25
(square 7) => 49
```

`Let`, on the other hand, is mainly used to name parts of more complex expressions in order to increase efficiency and/or readabiliy. For instance, the following procedure searches an environment of the form that is used in `let`:

```
(define (value-of sym env)
  (cond ((null? env) #f)
        ((eq? sym (car (car env))) (car env))
        (else (value-of sym (cdr env)))))
```

When the search for the symbol *sym* in the environment *env* is successful,

it returns the binding of the given symbol and otherwise it returns `#f` to signal failure:

```
(value-of 'b '((a 1) (b 2) (c 3))) => (b 2)
(value-of 'x '((a 1) (b 2) (c 3))) => #f
```

If you wanted to write a procedure that evaluates to the value of a symbol if the symbol is in *env* and to the symbol itself otherwise, you would have to call *value-of* twice:

```
(define (substitute sym env)
  (if (value-of sym env)
      (cadr (value-of sym env))
      sym))
```

The second application of *value-of* can be saved by using `let`:

```
(define (substitute sym env)
  (let ((val (value-of sym env)))
    (if val (cadr val) sym)))
```

Instead of evaluating `(value-of sym env)` twice in case of a match, this version of *substitute* binds the result returned by *value-of* to *val* and uses the variable in the place of the application of *value-of*. This version is both more readable, because `val` is easier to read than `(value-of sym env)`, and more efficient, because *value-of* is only called once, no matter whether the given symbol exists in the environment or not.

BTW: in a real Scheme program, you would use the built-in `assoc` procedure rather than coding `value-of` yourself. `Assoc` has the same cost as `value-of`, though, so the above example remains valid.

Here is another application for `let` which demonstrates how to simplify an expression. Given the coordinates of two points on a grid, the distance between the points is to be calculated. Using Pythagoras' theorem, the distance is computed by:

square-root (distance-on-x-axis^2 *+ distance-on-y-axis*^{2} *)*

If the coordinates of one point are *(x,y)* and the coordinates of the other point are *(x2,y2)*, the distance on the x-axis is $|x2-x|$ and the distance on the y-axis is $|y2-y|$, where $|x|$ denotes the absolute value of *x*. So the procedure to compute the distance in Scheme looks like this:

```
(lambda (x y x2 y2)
  (sqrt (+ (* (abs (- y2 y))
              (abs (- y2 y)))
           (* (abs (- x2 x))
              (abs (- x2 x))))))
```

Although indentation even makes the formula readable, it can be simplified by extracting the distances on the individual axes using `let`:

```
(define (distance-between-points x y x2 y2)
  (let ((dx (abs (- x2 x)))
        (dy (abs (- y2 y))))
    (sqrt (+ (* dx dx)
             (* dy dy)))))
```

In applications of Scheme procedures, the *order of evaluation* is unspecified. This means that in a procedure application like

```
(+ (* 1 2) (* 3 4) (* 5 6))
```

the arguments `(* 1 2)`, `(* 3 4)`, and `(* 5 6)` may be evaluated in *any* order before they are passed to `+`. Because `let` is equivalent to the application of a (lambda) function, the same is valid for `let`:

```
(let ((a (* 1 2))
      (b (* 3 4))
      (c (* 5 6)))
  (+ a b c))
```

When creating local contexts using `let`, it is important that the definitions of the local values be independent (which they are in the above example). No value specified in the environment of a `let` may refer to a variable defined in the same `let`. This is because `let` first evaluates *all* values in its environment and *then* binds the values to symbols. The above expression would first evaluate `(* 1 2)`, `(* 3 4)`, and `(* 5 6)` in some unspecific order and only after evaluating all of these expressions, it would bind *a* to 2, *b* to 12, and *c* to 30. Dependencies between definitions of the same `let` will lead to undesired results:

```
(let ((a 5)
      (b (+ 1 a))
      (c (+ 1 b)))
  c)
=> bottom
```

Because *a* is bound to 5 *after* the evaluation of all values of the environment, *a* is not bound at all when evaluating `(+ 1 a)`. The same is valid for the *b* in the value of *c*. The desired effect can be achieved, though, by nesting `let`:

```
(let ((a 5))
  (let ((b (+ 1 a)))
    (let ((c (+ 1 b)))
      c)))
=> 7
```

In this example, the value to be bound to *b* is computed *inside of* the context in which *a* is bound to 5, and *c* is bound *inside of* the context in which *b* is bound to 6. Because nesting applications of `let` may look a bit cumbersome, Scheme provides a binding construct which evaluates and binds values in sequence:

```
(let* ((a 5)
       (b (+ 1 a))
       (c (+ 1 b)))
  c)
=> 7
```

The star in `let*` indicates that "*the construct does something sequentially*". `Let*` evaluates the first value and binds it to a symbol, thereby creating a new context. In this context it evaluates the second value and binds it to a symbol, creating yet another context, and so on. The expression forming the value of a symbol may refer to all variables created *before* it in the same `let*`. The following expressions are equivalent:

```
(let ((v1 a1))          =  (let* ((v1 a1)
  (let ((v2 a2))                 (v2 a2)
   ...                             ...
    (let ((vn an))                (vn an))
      body)))                body)
```

1.7.1 Binding Recursive Procedures

In the code so far `letrec` was used to bind procedures and `let` was used to bind all other values. However, this is not really the difference between these two binding constructs. `Letrec` can be used to bind values, too:

```
(letrec ((a 17)
         (b 4))
  (+ a b))
=> 21
```

and `let` can be used to bind *trivial* procedures:

```
(let ((square (lambda (x) (* x x))))
  (square 7))
=> 49
```

The order of evaluation of the values of `letrec` is unspecified, just as the order of evaluation in `let`. So what *is* the difference between `let` and `letrec`? A minor hint was already given in a sentence above: `let` can only be used to bind *trivial* procedures. In this case "*trivial*" means "*not using recursion*", and this is exactly what `letrec` is for: binding recursive procedures. In case you wonder why `let` cannot be used for this, take a look at the following expression:

```
(let ((down
        (lambda (x)
          (if (zero? x)
              0
              (down (- x 1))))))
  (down 5))
=> bottom
```

When you try this example, your Scheme system will complain about *down* being unbound. How can this be?

First the value to be assigned to *down* is evaluated. This involves the application of `lambda` which returns a closure. And this is exactly the problem. Lambda closes over *down* **before** *down* is bound to the procedure. In the resulting closure, *down* is unbound, and when the closure is applied to a non-zero value, an error occurs. (When the closure is applied to zero, no recursion takes place and 0 is returned, just as expected.)

What `letrec` does after binding values to symbols in the same way as `let` does is to fix recursive references. It even fixes indirect recursive references which are created by *mutually recursive* expressions like these:

```
(letrec
  ((even-p
     (lambda (x)
```

```
        (if (zero? x) #t (odd-p (- x 1)))))
    (odd-p
      (lambda (x)
        (if (zero? x) #f (even-p (- x 1))))))
  (cons (even-p 5) (odd-p 5)))
=> (#f . #t)
```

To fix recursive bindings, `letrec` needs a construct that is not purely functional, because it changes the value of a variable. This construct is the `set!` pseudo function. The trailing "`!`" in the name `set!` indicates that it does something that requires special attention. Constructs ending with "`!`" are used with care in Scheme.

`Set!` changes the value associated with a variable:

```
(define foo 0)
foo => 0
(set! foo 1)
foo => 1
```

It differs from `define` in two ways:

– `define` introduces a new variable, which `set!` does not;
– `set!` can be used anywhere, `define` only in specific contexts.

Standard Scheme requires a variable to be `define`d before you can `set!` it, but some Scheme environments do not enforce this rule.

How does `set!` help `letrec` define recursive procedures? Like `let*`, `letrec` can be re-written to `let` (and `set!`):

```
(letrec ((down
           (lambda (x)
             (if (zero? x) 0 (down (- x 1))))))
  (down 5))
```

becomes

```
(let ((down #f))
  (let ((t0 (lambda (x)
              (if (zero? x) 0 (down (- x 1))))))
    (set! down t0)
    (down 5)))
```

The outer `let` binds the variable *down* to something unimportant. The

only purpose of this definition is to introduce the procedure name *before* the inner `let` is entered. In the inner `let`, `lambda` closes over the binding of *down* to an unimportant value and then `let` binds the resulting procedure to the temporary variable *t0*. In the body of the inner `let`, the binding of *down* to `#f` is still visible. `Set!` changes its value to the procedure bound to *t0*. This mutation also affects the value that `lambda` closed over, because the scope of the outer `let` is still in effect. The resulting structure is recursive:

```
down = (lambda (x)
         (if (zero? x) 0 (down (- x 1))))
```

In general, `letrec` can be transformed to `let` in the following way:

```
(letrec ((v1 p1)  =  (let ((v1 #f) ... (vn #f))
         ...            (let ((t1 p1)
         (vn pn))             ...
  body)                       (tn pn))
                        (set! v1 t1)
                        ...
                        (set! vn tn)
                        body))
```

`Set!` and `letrec` are the only Scheme constructs discussed in this book which can create cyclic structures. If you think that you can create a recursive structure in a purely applicative subset of Scheme without using constructs like `letrec` or `set!`, I really would like to see how it works (but do not try too hard, it may turn out to be impossible).

1.8 Lists, Pairs, and History

A good definition of the list is still lacking. To give one, another type that already has occurred in this book will be explained first: the *pair*. A pair is what you get when you *cons* together two *atoms*. An atom, as the name suggests, is a type that cannot be decomposed. (“Atomos” is the Greek word for “indivisible”.) A Scheme object that cannot be decomposed is said to be “atomic”. The following objects are atomic:

- Symbols
- Booleans
- Procedures
- Numbers

– Chars
– Strings
– The empty list

In the previous sections, the second argument of `cons` has always been a list. So what happens when `cons` is applied to two atoms? Here is the answer:

```
(cons 'a 'b) => (a . b)
```

The two objects are ''glued together'' and form something that is called a *dotted pair*. To understand how his works, it is probably best to disclose some internal details. The `cons` procedure actually creates a special type of object that is sometimes referred to as an *object of the type cons*. Its only purpose is to glue together two objects. The dot of a dotted pair may be considered to be the external representation of a the cons object itself.

What is particularly interesting is what happens when the *empty list* is consed to an atom:

```
(cons 'a ()) => (a)
```

You might have expected an object of the form `(a . ())` but what you get is a list with a single member. The empty list seems to have vanished, but it has not. In fact a single-element list *is* a pair whose cdr part is the empty list:

```
'(a . ())  =  (a)
```

Gathering all information about lists that have been given so far, the following rules for constructing lists can be devised:

– `()` is a list (obviously);
– `(Cons x '())` is a list for any object *x*;
– If *y* is a list (and *x* is any object), `(cons x y)` is a list.

The second case is in fact redundant, because `()` is a list and so case two is just a variant of case three. This method of constructing lists was used in the self-made *append2* procedure at the beginning of this book. *Append2* created some nested `cons`es in order to create a new list:

```
(cons 'a (cons 'b '(c d))) => (a b c d)
```

Every list can be written in dotted pair notation. For example

```
(a b c)  =  (a b c . ())
         =  (a b . (c . ()))
         =  (a . (b . (c . ())))
```

The first line is particularly interesting, because it shows that the cdr part of the last element of a list is always `()`. This is why `null?` can be used to detect the end of a list. (Other dialects of LISP call the empty list `NIL` which means "*Not In List*".) Above illustration also makes clear why taking the cdr part of a list results in a list. Dotted pair notation also can be used to show why consing a list to a list does not append the lists:

```
(cons '(a b) '(c d))
=> ((a b) . (c d))
=  ((a b) c d)
```

Above definition for the list implies that the last element of each list must be `()`. Is it possible, though, to append an atom to a list? Let us see:

```
(append '(a b c) 'd) => (a b c . d)
```

The result of this application is called an *improper list*. It is called "improper" *just because* it does not end with `()`, which every *proper* list must do. Because the last member of an improper list is not the empty list, the external representation of such a list includes a dot marking its last cons. You can create improper lists not only by appending an atom to a list, but also by just typing them in:

```
'(a b c . d) => (a b c . d)
```

However, most procedures of Scheme expect proper lists and reduce to bottom when an improper list is passed to them:

```
(reverse '(a b . c)) => bottom
(append '(a b . c) '(d)) => bottom
```

Improper lists are not useless, though. Even the Scheme language itself makes use of them. The details will be discussed in a later section.

1.8.1 Association Lists

A list that consists of pairs exclusively is called an *association list* (or *alist*). Alists are frequently used to form environments. Some Scheme implementations even use them to store their own environments, like the lexical environments of closures.

Each pair of an alist is called an ‘‘association’’. The car part of an association is called the ‘‘key’’ and the cdr part is called the ‘‘value’’ of that association:

```
((key_1 . value_1) ... (key_n . value_n))
```

To retrieve an association from an alist, the `assoc` procedure is used. It returns the first association containing the given key. In case no matching association is found, it returns `#f`:

```
(define alist '((a . 1) (b . 2) (c . 3) (b . 4)))
(assoc 'b alist) => (b . 2)
(assoc 'x alist) => #f
```

Although `assoc` always returns a true value when it finds a matching association and false if it does not find one, it is not a predicate, because it returns any true value in case of success and not always `#t`.

Note that associations created by alists are independent from bindings of symbols. Although the above alist associates symbols with values, it does not change the bindings of any symbols. The symbols themselves serve as keys in alists:

```
(define x 'foo)
x => foo
(assoc 'x '((x . bar))) => (x . bar)
x => foo
```

The keys of alists are not really limited to symbols, but they can be of any type. There may even be different types of keys in the same alist:

```
(define alist '((#f . first)
                ("hello" . second)
                ((a b c) . third)))
      (assoc #f alist) => (#f . first)
 (assoc "hello" alist) => ("hello" . second)
(assoc '(a b c) alist) => ((a b c) . third)
```

1.8.2 Nested Lists

Another thing you might want to know about lists is how to access members of nested lists efficiently. Of course, the `car` and `cdr` procedures can be combined to achieve this, but nested applications of these procedures soon become a mess to read:

```
(define x '(define (square x) (* x x)))
(car x) => define
(car (cdr x)) => (square x)
(car (car (cdr x))) => square
(car (cdr (car (cdr x)))) => x
```

You also might want to know why Scheme still uses the names `car` and `cdr` (which are artifacts from the early LISP days of the 1950's) to refer to the heads and tails of lists. Other dialects of LISP use names like "first" and "rest" or "head" and "tail", so why does Scheme stick with the traditional names? The answer is simple: because they can easily be combined to form new names for procedures that access members of nested lists. For example, a combination of `car` and `cdr` is used to access the second element of a list:

```
(car (cdr '(a b c))) => b
```

If you take the "a" of `car` and the "d" of `cdr`, you can use these letters to create the name `cadr`. This is the name of the Scheme procedure for accessing the car of the cdr of a list (or the head of the tail of a list (or the first member of the rest of a list)):

```
(cadr '(a b c)) => b
```

Scheme provides built-in procedures for accessing up to four levels of nesting, such as

```
              (cddr '(a b c)) => (c)
             (caddr '(a b c)) => c
 (caddr '((a b) (c d) (e f))) => (e f)
(caaddr '((a b) (c d) (e f))) => e
(cadadr '((a b) (c d) (e f))) => d
```

Figuring out which member a procedure like `cadadr` produces is simple. Read the a's and the d's of the name backward and perform a `car` for each "a" and a `cdr` for each "d", as in the following example:

```
(cadadr '((a b) (c d) (e f)))  ; do a cdr
-> (cadar ((c d) (e f)))       ; do a car
-> (cadr (c d))                ; do a cdr
-> (car (d))                   ; do a car
=> d
```

Of course, these procedures are only useful when decomposing lists on the

fly. If you plan to use lists as data structures, you would give more descriptive names to procedures extracting specific members. For example, a date could be stored in a list of the form

```
(year month day)
```

but of course, you would not use the names `car`, `cadr` and `caddr` to refer to the single fields of a date object. You would define accessor procedures like these:

```
(define (year date) (car date))
(define (month date) (cadr date))
(define (day date) (caddr date))
```

Of course, `cadr` and friends as well as custom names for accessing parts of structured lists are only suitable for handling rather flat and short lists. More complex list structures are typically handled by ''traversing'' them. The built-in `length` procedure, for instance, traverses a flat list and counts its members, resulting in the length of that list:

```
(length '(a b c)) => 3
(length '(a (b (c) d) (e (f (g))) h)) => 4
```

The (also built-in) `list-ref` procedure produces the *n*th member of a list, where the first member is at position zero:

```
(list-ref '(a b c) 0) => a
(list-ref '(a b c) 2) => c
(list-ref '(a b c) 3) => bottom
```

When no appropriate procedure is available, a specialized one must be coded. For example, Scheme has no procedure for computing the *depth* of an object (probably because it is not terribly useful). However, writing a depth procedure may serve as an interesting exercise. The depth of an object is the maximum number of times that `car` can be applied to that object. The `car`s do not have to be in a row, but they can be mixed with any number of `cdr`s. For example,

```
          (depth 'x) => 0
        (depth '(x)) => 1
      (depth '((x))) => 2
(depth '(u (v (x)))) => 3
```

So the depth of each member of a given list has to be computed and then

the maximum of the results has to be chosen. To compute the depth of each sublist, depth has to recurse. The trivial case is simple: each atomic object has a depth of zero. The depth of non-atomic objects is the depth of the deepest object contained in it plus one:

```
(define (depth x)
  (if (not (pair? x))
      0
      (+ 1 (deepest-sublist x))))
```

The depth of the deeptest sublist is computed by applying *depth* to each sublist, collecting the results, and then returning the maximum of the results. To do so, a procedure for finding the maximum of a list is required:

```
(define (max-list a)
  (cond ((null? (cdr a)) (car a))
        ((> (car a) (cadr a))
          (max-list (cons (car a) (cddr a))))
        (else (max-list (cdr a)))))
```

(Note: the > procedure, which makes sure that its arguments are in numerically strict descending order, is a predicate, even if its name does not end with a question mark.)

The only procedure that is left to implement is *deepest-sublist*. This procedure has to transform a list of objects into a list of numbers, where each number indicates the depth of the object it replaces. After transforming the list, it is passed to *max-list* to extract the greatest depth found in the list. Here is the code:

```
(define (deepest-sublist x)
  (letrec
    ((transform
       (lambda (in out)
         (cond ((null? in) out)
               (else (transform (cdr in)
                                (cons (depth (car in))
                                      out)))))))
    (max-list (transform x '()))))
```

When looking at the *transform* procedure, you might notice an opportunity for an optimization in the following expression.

```
(cons (depth (car in)) out)
```

Got it? Before doing the next iteration in *transform*, the depth of the current sublist is consed to *out*. If the maximum depth found so far was bound to *out* instead of the list, the new maximum depth could be computed right at this point. In this case, the *max-list* procedure could be optimized away altogether. Of course, *transform* and *out* would have to get more descriptive names, too:

```
(define (deepest-sublist x)
  (letrec
    ((max-depth
       (lambda (in max-so-far)
         (cond ((null? in) max-so-far)
               (else (max-depth (cdr in)
                                (max (depth (car in))
                                     max-so-far)))))))
    (max-depth x 0)))
```

Optimizations like this spring to mind quite frequently. The first program rarely is the optimal solution (if such a thing exists anyway). The most important goal is to make an algorithm work at all. Even the first version of *depth* worked fine, and there is barely a (non-trivial) program that cannot be optimized in one or another way. Do you think that *depth* cannot be expressed in an even simpler way? We will see.

1.8.3 A Historical Note

The names `car` and `cdr` date back to the very first implementation of LISP in the late 1950's. This version of LISP ran on an ''IBM Type 704 Electronic Data Processing Machine'', which was a vacuum tube-based computer.

The memory of that computer consisted of up to 32768 machine words of magnetic core, so 15 bits were sufficient to address each word of storage. A machine word on the IBM 704 had 36 bits, so a complete cons object could be stored in a single word or register.

The registers of the 704 had four parts named ''prefix'', ''address'', ''tag'', and ''decrement''. The address and decrement parts were used to store addresses, so they had a size of 15 bits each. Hence they were used to store the references to the car and cdr part of a cons cell.

The routine for extracting the head of a cons cell was called `CAR`, which

was short for "*Content of Address part of Register*". The routine to extract the tail was called `CDR` for "*Content of Decrement part of Register*".

Although the technical details vanished when LISP was being ported to other machines, the names `car` and `cdr` persisted, and most dialects of LISP still keep them today, because they can easily be combined to form new procedure names like `cadr`.

2 Less Basic Scheme Programming

2.1 Variable Argument Procedures

Many languages provide a means of defining procedures that take a variable number of arguments, and Scheme is among them. Normally each variable is bound to one argument during a procedure call. For example, the following *max* procedure takes two arguments and returns the greater one of them:

```
(define (max a b)
  (if (> a b) a b))
```

When *max* is called, its argument *a* is bound to the first argument and *b* is bound to its second argument:

```
(max 34 43)
-> (if (> 34 43) 34 43)
```

However, the real `max` procedure of Scheme is a *variadic* procedure, which means that it accepts any positive number of arguments:

```
(max 5 1 3 8 9 7 2 6 4) => 9
```

In the previous section, the *max-list* procedure was defined. This procedure extracted the greatest member from a *list* of values. It had to be called with a single argument which was a list of the numbers to process:

```
(max-list '(5 1 3 8 9 7 2 6 4)) => 9
```

So the algorithm is already there. The only problem left is to stop Scheme from decomposing the list of arguments when calling a procedure. This is where improper lists come into play. A procedure with an improper argument list takes any number of arguments, but *at least* as many as it has variables before the dot. Here is an example:

```
(define (fv2 a b . c) c)
```

This procedure accepts *at least two* arguments. Its first argument will be bound to *a* and the second one to *b*. The list of remaining arguments will be bound to *c*:

```
(fv2 'foo 'bar 'baz)    => (baz)
(fv2 'a 'b 'c 'd 'e 'f) => (c d e f)
```

If the procedure has exactly two members, the empty list is bound to *c*:

```
(fv2 'a 'b) => ()
```

Passing any smaller number of arguments to *fv2* results in an error:

```
(fv2 'a) => bottom
(fv2) => bottom
```

The *max* procedure should expect at least one member, because applying *max* to zero members would not make any sense. Here is a procedure that can be wrapped around *max-list* in order to create a *max* procedure taking a variable number of arguments but at least one:

```
(define (max x . a)
  (max-list (cons x a)))
```

The *x* before the dot makes Scheme report an error when less than one argument is passed to *max*. However, the *x* has to be consed to the list holding the rest of arguments before passing the list to *max-list*. This could be avoided by making *max* accept zero or more arguments. In this case, though, the *max* procedure itself would have to make sure that at least one argument was passed to it:

```
(define (max . a)
  (if (null? a)
      (bottom "max requires at least one argument")
      (max-list a)))
```

This definition has no symbols between the procedure name and the dot of the argument list, so it binds the *entire* list of arguments to *a* without doing any decomposition. When zero arguments are passed to this version of *max*, the `bottom` procedure is used to abort the computation and report an error.

Note that the `bottom` procedure does not really exist in Scheme. Because it does not exist, Scheme will print an error message and abort the current computation when it encounters an application of `bottom`, which is exactly what `bottom` is intended to do. If your Scheme system offers a (non-standard) procedure like `error` or `wrong`, you may prefer to use it in the place of `bottom`.

Because (non-primitive) procedures are created using `lambda`, there must be a way to create variadic lambda functions, too. Indeed this is possible:

```
((lambda (a . b) b) 'foo 'bar) => (bar)
((lambda (a . b) b) 'a 'b 'c)  => (b c)
((lambda (a . b) b) 'foo)      => ()
```

Using `lambda`, procedures that are local to a `letrec` can be made variadic, too. It is even possible to create procedures accepting zero or more arguments using `lambda`. A pair obviously cannot be used to achieve this, because a pair with no car part would not be a pair. So how does it work? Decomposing the signatures of some procedures gives the answer. The *signature* of a procedure is a list containing the name and the variables of a procedure, as in

```
(define (f2 a b . c) c)
```

The cdr part of a signature is the argument list of a procedure, e.g.:

```
; Definition           Signature    Arguments
(define (f x) y)       (f x)        (x)
(define (f x . y) y)   (f x . y)    (x . y)
(define (f . x) x)     (f . x)      x
```

So the argument list of a variadic procedure with zero or more arguments is not a list at all, but a single variable. This variable will be associated with the entire list of arguments when the procedure is called:

```
((lambda x x) 'a 'b 'c) => (a b c)
((lambda x x) 'foo)     => (foo)
((lambda x x))          => ()
```

BTW: `(Lambda x x)` happens to be the code of a very useful Scheme procedure named `list`. All it does is to evaluate to the list of arguments passed to it. Because Scheme evaluates arguments *before* passing them to procedures, the resulting list will be a list of normal forms, which is different from specifying a literal list:

```
    '((+ 5 7) (* 3 4)) => ((+ 5 7) (* 3 4))
(list (+ 5 7) (* 3 4)) => (12 12)
```

Because variadic arguments are passed as lists, they cannot simply be passed to other variadic procedures. For example, the following attempt ro write a procedure that extracts the limits of a list (its least and its greatest member) fails:

```
((lambda a (list (min a) (max a))) x) => bottom
```

This expression reduces to bottom for *any* value of *x*, because a list is passed to `min` and to `max` while both of these procedures expect numbers as their arguments. If `min` and `max` were two-argument procedures, this would be simple:

```
(lambda a
  (list (min (car a) (cadr a))
        (max (car a) (cadr a))))
```

This approach decomposes the variable argument list using `car` and `cadr` and then passes the extracted arguments to `min` and `max`. This method works only if the number of arguments is known in advance, though.

The program would be much nicer and simpler, if the variable argument list *itself* could just be passed to `min` and `max`. This is the point where the `apply` procedure comes into play:

```
(define (limits . a)
  (list (apply min a)
        (apply max a)))
```

`(Apply f a)` applies a procedure *f* to a list of arguments *a*:

```
(apply f '(1 2 3)) -> (f 1 2 3)
```

Of course the number of members of the list must match the number of arguments that the procedure expects:

```
(apply cons '(a)) -> (cons 'a) => bottom
```

`Apply` is an ordinary procedure. Its arguments are passed to it using *call by value*. A procedure is called "by value", if all arguments of that procedure are evaluated *before* they are passed to the procedure. In Scheme *all* procedures are called by value. Pseudo functions, on the other hand, are applied "by name". Passing arguments to a function using *call by name* means that the arguments are *not* evaluated before they are passed to the function. For example, `quote` is called by name:

```
(quote (+ 2 3)) => (+ 2 3)
```

while the `list` procedure is called by value:

```
(list (+ 2 3)) => (5)
```

`(Apply f a)` *receives* its arguments by value, but applies *f* to the

members of *a* using call by name:

```
(apply cons '(a b))
-> (cons a b)
=> (a . b)
```

If `(apply f a)` would pass the members of *a* using call by value, they would be evaluated twice: once when calling `apply` and another time when applying *f*. In the above example this would mean that the *values* of *a* and *b* would be passed to `cond`, resulting in *bottom* if *a* or *b* was unbound. If you actually wanted to pass the values of some variables *a*, *b*, and *c* to a procedure using apply, you would use `list`:

```
(apply + (list a b c))
```

but then again, this would be equivalent to using the procedure being applied directly:

```
(apply + (list a b c))  =  (+ a b c)
```

BTW: The *compose* procedure introduced earlier in this book can be improved by using `apply`, too. The version introduced earlier can only compose unary procedures (procedures of one argument). The improved version accepts a procedure of any arity (any number of arguments) in the last position:

```
(define (compose f g)
  (lambda x (f (apply g x))))
```

2.2 Identity and Equality

In Scheme, there are many procedures for testing for equality, but there is only one procedure for testing for *identity*. As outlined earlier, only appearances of *the same* object can be identical. The definition of ''equality'' is a much broader one.

There are only three types of objects that can be tested for identity: the symbol, the boolean, and the empty list. It would be incorrect to state that ''*two symbols are identical, if they have the same name*'', because identity is a relation that only exists between an object and itself. If there are two or more symbols, they cannot be identical. However, the name of one symbol can be written down any number of times, like this:

```
foo  foo  foo  foo  foo
```

In this case, the *name of the symbol* exists five times, but the symbol is still unique. Sometimes the name of a symbol is referred to as a symbol, which is confusing with regard to identity. It is better to state that all these names *refer* to the same symbol. The `eq?` procedure tests whether two expressions reduce to the same symbol:

```
(eq? 'foo 'foo) => #t
(eq? 'foo 'bar) => #f
(eq? 'foo (car '(foo))) => #t
```

The empty list `()` shares the property of uniqueness with symbols. There is only one empty list in Scheme (which is why it is frequently referred to as *the* empty list), so `eq?` can be applied to it safely:

```
 (eq? '() '()) => #t
(eq? 'foo '()) => #f
```

The truth literals `#t` and `#f` are unique, too:

```
(eq? #f #f) => #t
(eq? #t #t) => #t
```

If *one* of the two arguments of `eq?` is neither a symbol nor an empty list nor a boolean, the result is negative:

```
  (eq? 'foo 5) => #f
  (eq? #t #\x) => #f
(eq? '() '(y)) => #f
```

So `eq?` performs some implied type checking. If the types of its arguments do not match, it always returns falsity.

If *both* arguments of `eq?` are neither symbols nor empty lists nor a booleans, the result is *undefined*:

```
      (eq? 5 5) => bottom
  (eq? #\a #\a) => bottom
(eq? '(x) '(x)) => bottom
```

Note that *bottom* really means "*undefined*" in this case. Some implementations of Scheme may return truth when `eq?` is applied to 5 and 5, some may return falsity, some may return truth most of the time and falsity another time. **Just do not use** `eq?` **to compare with anything but symbols**

and booleans, and you are on the safe side. Even to compare with `()`, `null?` should be preferred.

There are lots of other types in Scheme which can be compared. However, these types are compared in terms of equality or equivalence. There is a whole bunch of different procedures for comparing different types of objects. For example, these procedures are used to compare numbers:

```
<   <=   =   >   >=
```

These are predicates, but because their names consist of special characters exclusively, they have no question marks attached. The meanings of these symbols are the same as in mathematics or in many other programming languages: < means "*less than*", >= means "*greater than or equal to*", and = tests for numeric equivalence. There is no predicate to test whether a sequence of values is *not* equivalent, though. The reason will become clear soon.

The *numeric predicates* of Scheme are variadic procedures that take at least two arguments, and

```
(< a b c d)
```

actually tests whether *a* is less than *b and b* is less than *c and c* is less than *d*. In other words, < tests whether its arguments are in monotonically ascending order. The <= predicate tests whether its arguments are in monotonically non-descending order, and = tests whether its arguments are all equivalent.

Given these definitions, what is a hypothetical *not*= (not-equivalent) predicate to do? To be in line with the above definitions

```
(not= a b c)
```

would mean that *a* is not equal to *b and b* is not equal to *c*. In this case, though, *not*= would have a meaning hat differs from

```
(not (= a b c))
```

The latter would denote that *a* is not equal to *b* **or** *b* is not equal to *c*.

Because each of the above definitions of negative equivalence is flawed in one or another way, the Scheme standard does the only right thing to do: it does not include such a definition at all.

There are predicates similar to = and friends that are used to compare chars and strings. They work in the same way as the numeric predicates, but compare chars and strings respectively. These procedures are not *required* by the standard to accept more than two arguments, but implementations may choose to make them variadic. The names of string and *char predicates* are formed by attaching the prefixes ''string'' or ''char'' and appending a question mark:

```
numeric  char      string
<        char<?    string<?
<=       char<=?   string<=?
=        char=?    string=?
>        char>?    string>?
>=       char>=?   string>=?
```

A char *a* is considered ''less than'' a char *b*, if

- they are both upper case letters and *a* comes before *b* in the alphabet;
- they are both lower case letters and *a* comes before *b* in the alphabet;
- they are both digits and the value of *a* is less than the value of *b*.

The definitions of the other char predicates can be derived from the above rules using basic logic:

```
(char>? a b)   =  (char<? b a)
(char<=? a b)  =  (not (char>? a b))
etc
```

When comparing letters, only the alphabetic part of the ASCII definition is taken into consideration. The order of characters other than letters and decimal digits (''special'' characters) is undefined, and so are the relations between letters and digits, letters and special characters, and digits and special characters:

```
(char>? #\x #\%) => bottom
(char>? #\+ #\-) => bottom
(char>? #\+ #\0) => bottom
```

The char equivalence predicate has no such limitations, though:

```
(char=? #\a #\0) => #f
(char=? #\* #\*) => #t
```

When the name of a char predicate begins with ''char-ci'' instead of

''char'', then the predicate folds the case of its arguments before comparing them. The abbreviation ''ci'' means *case insensitive*. For example,

```
(char<? #\a #\b) => #t
(char<? #\a #\B) => bottom
```

but

```
(char-ci<? #\a #\B) => #t
```

Strings are sequences of chars, so the *string predicates* basically use the char predicates to compare the individual characters of two strings. `String=?` evaluates to truth, if the strings passed to it contain equal characters at corresponding positions:

```
(string=? "abcd" "abcd") => #t
(string=? "dcba" "abcd") => #f
```

The other predicates skip over all characters that are equal in both strings and then apply the corresponding char predicate to the first characters that do not match, returning its result:

```
(string<? "abcd" "abcz") => #t
(string>? "abcd" "abca") => #t
```

If two strings have different lengths and contain equal characters up to the end of shorter string, the shorter string is considered ''less than'' the longer string:

```
(string<? "abc" "abcd") => #t
```

Like the char predicates, all of the string predicates have case insensitive counterparts called `string-ci=?`, `string-ci<?`, etc. They work in the same way as the case sensitive string predicates, but apply the corresponding case insensitive char predicates instead.

Given all the different types of equivalence described in this section, a general predicate for testing all kinds of objects for equality would be desirable. Such a predicate is discussed in the following section.

2.2.1 A More General Form of Equality

The predicates discussed in the previous section express identity and equivalence. Identity, the sameness of two objects, already has been dis-

cussed before. Equivalence expresses that two objects have "the same value". It is typically applied to objects on which an ordering can be imposed, like numbers, characters, or strings.

Scheme also provides a more general equivalence predicate that can be used to compare objects of different types: `eqv?` is a less strict variant of `eq?` that can be used to compare numbers and characters:

```
    (eqv? 'x 'x) => #t
    (eqv? #f #f) => #t
  (eqv? '() '()) => #t
  (eqv? #\Z #\Z) => #t
(eqv? -123 -123) => #t
```

For each pair of arguments for which `eq?` returns truth, `eqv?` returns truth as well. `Eqv?` performs the same implied type checking as `eq?`:

```
 (eqv? 5 "x") => #f
(eqv? 7 '(x)) => #f
```

But even `eqv?` cannot be used to compare two strings or pairs:

```
(eqv? "abc" "abc") => bottom
  (eqv? '(x) '(x)) => bottom
```

While strings can be compared using `string=?` or `string-ci=?`, none of the procedures discussed so far could compare pairs.

Pairs have no order, so equivalence does not apply to them. Because the application of `eq?` to pairs is undefined, identity does not help either, which leaves *equality*. To give a first impression of general equality, only symbols and lists of symbols will be taken into consideration. Two lists of symbols are equal, if they contain the same symbol at corresponding positions. Since the list is only a special case of the pair, though, an approach based on the pair would be more promising. Here is a first definition:

Two objects are equal, if

- they are the same symbol;
- they are both pairs and contain equal car and cdr parts.

The conversion of this definition into code is quite straight-forward:

```
(define (eql? a b)
  (if (pair? a)
```

```
(if (pair? b)
    (if (eql? (car a) (car b))
        (eql? (cdr a) (cdr b))
        #f)
    #f)
(eq? a b)))
```

The condition "*a and b are both pairs and contain equal car and cdr parts*" is implemented in a rather awkward way in this procedure by using three nested applications of `if`. Scheme provides a much better way of expressing logical conjunctions. The expression "*a* is a pair *and b* is a pair" can be written as:

```
(and (pair? a) (pair? b))
```

The `and` pseudo function does not just implement a logical "*and*", though. In fact it implements conditional evaluation in a similar way as `cond` or `if`. `And` evaluates its arguments in sequence until either an argument reduces to `#f` or it runs out of arguments. As soon as one of its arguments evaluates to `#f`, it stops instantly and returns falsity, ignoring the remaining arguments, if any. Therefore, it is safe to write code like this:

```
(and (pair? a)
     (pair? (car a))
     (caar a))
```

If *a* is not a pair, `and` returns `#f` immediately and the expressions `(pair? (car a))` and `(caar a)` are not evaluated at all. Only if *a* already turned out to be a pair, the next expression is evaluated, and only if `(car a)` is a pair, too, the final expression is reduced. In this case, the value of the final expression is the value of the entire application of `and`. By skipping the remaining expressions as soon as a predicate could not be satisfied, the above application of `and` protects `car` from taking the car part of a non-pair.

Formally `and` can be re-written using `if` in the following way:

```
(and a b)     =  (if a b #f)
(and a b c)   =  (if a (if b c #f) #f)
etc
```

Using `and`, the above *eql?* predicate can be simplified significantly:

```
(define (eql? a b)
  (if (and (pair? a)
           (pair? b))
      (and (eql? (car a) (car b))
           (eql? (cdr a) (cdr b)))
      (eq? a b)))
```

Based on the *eql?* procedure, a general predicate for testing the equality or equivalence or identity of any two objects can be devised. Such a predicate could be used to compare any type of objects without ever reducing to bottom (with one exception that will be outlined later). In order to compare *any* objects, some *type checking* has to be added. Using `and` makes this a piece of cake:

```
(define (equal? a b)
  (cond ((number? a) (and (number? b) (= a b)))
        ((char? a)   (and (char? b) (char=? a b)))
        ((string? a) (and (string? b) (string=? a b)))
        ((and (pair? a) (pair? b))
          (and (equal? (car a) (car b))
               (equal? (cdr a) (cdr b))))
        ((or (procedure? a) (procedure? b)) (bottom))
        (else (eq? a b))))
```

`Equal?` uses *type predicates* like `char?` and `number?` to find out what type an object has. A type predicate returns truth, if its argument has a given type. Type predicates allow `equal?` to choose the proper procedure for comparing the given objects. For example, the clause

```
((number? a)  (and (number? b) (= a b)))
```

is selected, if the argument *a* is numeric. In the body of the clause, `(= a b)` is only evaluated if *b* is also a number. If the types of *a* and *b* are not equal, `#f` is returned. `And` protects = from improper types. The same method is applied to the other types, so specialized predicates are always applied to the matching types. The clause handling pairs is based on the *eql?* procedure. The default clause handles symbols, booleans, and empty lists, which are checked for identity.

The `equal?` procedure can be used to check numbers, strings, chars, booleans, lists of any objects and any combination of these for equality, and it always delivers a meaningful result:

```
        (equal? 'foo 'foo) => #t
          (equal? 123 123) => #t
(equal? '(a (12) (b . c))
        '(a (12) (b . c))) => #t
 (equal? '(a . b) (a . c)) => #f
```

Equality can be considered a lax form of identity (every object is equal to itself), so `equal?` delivers truth when appearances of the same object are passed to it. Because `equal?` is such a versatile procedure, is is part of the Scheme standard.

One clause of `equal?` deserves a second glance for two reasons:

```
((or (procedure? a) (procedure? b))  (bottom))
```

One reason is that it employs the `or` pseudo function which has not been discussed yet, and the other reason is that is reduces to `(bottom)` without any further checking when a procedure is passed to `equal?`.

`Or` works in a similar way as `and`, implementing both the logical *or* and conditional evaluation. However, `or` evaluates its arguments in sequence until one of them evaluates to truth. As soon as an argument reduces to truth, `or` evaluates to that argument and ignores the remaining arguments. When all but the last argument reduce to falsity, it evaluates to the normal form of the last argument. `Or` can be re-written using `cond` like this:

```
(or a b)    =  (cond (a a) (else b))
(or a b c)  =  (cond (a a) (b b) (else c))
```

Applying a single argument to either `and` or `or` is equivalent to the argument alone:

```
(or a)   =  a
(and a)  =  a
```

Applying these pseudo functions to zero arguments yields the neutral element of the corresponding logic function:

```
(and) => #t
(or)  => #f
```

`#T` is the neutral element of the logical *and* operation because adding another `#t` to a sequence of expressions linked together using logical and does not change the value of that expression, just like adding a zero

does not change a sum and an additional factor of one does not change a product.

BTW: The arithmetic procedures `*` and `+` indeed deliver the neutral elements of the product and the sum respectively when they are applied to zero arguments.

The second question about the `equal?` predicate was why the equality of procedures is undefined. This question is perhaps best answered with another questions: "*Under which circumstances should two procedures be considered equal?*" Should they be equal, if their lambda expressions are equal? In this case, the two following procedures would be different, although they obviously do the same:

```
(lambda (x) x)
(lambda (a) a)
```

Or should they be considered equal, if they do the same? In this case, what would "doing the same" mean? Deliver the same values for each set of arguments? This would not only be impossible to check, because there is an infinite number of possible sets of arguments, it would also make quicksort and bubblesort equal, [2] because both of them sort a set of objects. And at this point, the topic of primitive procedures was not even touched. When should two primitive procedures be considered equal? When their machine codes are equal? What about implementations for different CPUs?

Defining the equality of procedures is a task that is far beyond the scope of a simple utility procedure. This is why the Scheme standard leaves this point undefined.

2.2.2 Using Identity and Equality

The rules for using `eq?`, `eqv?`, and `equal?` are simple. When in doubt, use `equal?`. When you are sure about the types to compare, use `eqv?`. When you are absolutely sure about the type, use the predicates specialized in that type. For example, use `=` to compare numbers and `string=?` or

2 Quicksort and bubblesort are both sorting procedures, but quicksort is known to be highly efficient while bubblesort is known to be one of the worst sorting algorithms around.

`string-ci=?` to compare strings. Of course, `equal?` can only check for equality, so if you want to examine the order of some objects, you have to use more specialized predicates like `<` and `char>?` anyway.

Only when dealing with symbols or truth values exclusively, `eq?` is the predicate of choice, because it expresses identity, which is what you actually want to check in this case, and `eq?` is much more efficient than `eqv?` or `equal?`. For instance, the logical *not* can be implemented quite efficiently like this:

```
(eq? predicate #f)
```

Whenever *predicate* evaluates to truth, this expression will reduce to falsity and vice versa. This actually is what the standard procedure `not` does.

When checking for the empty list, the `null?` predicate should be used. Using

```
(eq? x '())
```

would be fine from a technical point of view, but using `null?` probably expresses your intent more clearly.

There are some procedures which use `equal?` internally, like `assoc` and `member`. This is why `assoc` recognizes any type of object in the key fields of association lists. To search an alist containing symbolic keys exclusively, the `assq` procedure is much more efficient. To search an alist that contains numbers or characters as keys, the `assv` is most efficient. `Assq` uses `eq?` in the place of `equal?` and `assv` uses `eqv?`.

The `member` procedure also has counterparts that are based on `eq?` and `eqv?`. They are called `memq` and `memv`. Although they are highly efficient, their use is limited to the types covered by their equivalence predicates. For example, `memq` is limited to *flat lists of symbols* (or truth values):

```
(memq 'y '(x y z)) => (y z)
```

It cannot be used to search non-symbolic members of lists:

```
(memq '(c d) '((a b) (c d) (e f))) => bottom
```

In this case, you have to use `member` instead. Above rule of the thumb also applies here: when in doubt, use `assoc` and `member`.

Remember: Equality is a more general case than identity. Wherever `eq?`, `assq`, and `memq` return a non-`#f` result, `equal?`, `assoc`, and `member` will deliver the same result. The reverse is not true, though: (`==>` denotes a logical implication)

```
 (eq? a b) ==> (eqv? a b)
(eqv? a b) ==> (equal? a b)
```

but

```
(equal? a b) =/=> (eqv? a b)
  (eqv? a b) =/=> (eq? a b)
```

2.3 Higher Order Functions

A *higher order function* is a procedure that either takes a procedure as an argument or returns a procedure (or both). For instance, the *complement* and *compose* procedures introduced earlier in this book were higher order functions (HOFs). A classic among the HOFs is the *mapcar* procedure, which used to be part of early versions of LISP. *Mapcar* maps a procedure over a list. It is defined as follows:

```
(define (mapcar a f)
  (cond ((null? a) '())
        (else (cons (f (car a))
                    (mapcar (cdr a) f)))))
```

Using this procedure you can do quite a few useful things like computing the sum of squares of a set of numbers:

```
(define (sum-of-squares . x)
  (apply + (mapcar x (lambda (x) (* x x)))))
```

or find the member with the least absolute value in a list of numbers:

```
(define (least-abs x)
  (apply min (mapcar x abs)))
```

In both of these procedures, *mapcar* creates an argument list for a variadic procedure such as `+` or `min`. This combination can be seen frequently in Scheme programs. The following sample reduction of *sum-of-squares* illustrates how `apply` and *mapcar* interact:

```
(sum-of-squares 2 3 5 7)
-> (apply + (mapcar (2 3 5 7) (lambda (x) (* x x))))
-> (apply + (4 9 25 49))
-> (+ 4 9 25 49)
=> 87
```

The Scheme standard does not define *mapcar* but a much more powerful procedure called `map`. You may have observed that *mapcar* can only apply procedures of one argument to the members of a single list. `Map` eliminates this limitation. It can be applied to any number of lists. To allow for this, the procedure argument had to move to the first position. `Map` can do anything *mapcar* can do and more, like computing the inner product (the ''dot product'') of two vectors:

```
(define (v* a b) (apply + (map * a b)))
```

or the sum of two vectors:

```
(define (v+ a b) (map + a b))
```

or even the sum of *any* number of vectors:

```
(define (v+ . a) (apply map + a))
```

In the above defintion, `apply` takes three arguments rather than two. In fact `apply` itself is variadic. When more than two arguments are passed to it, the second through second-to-last arguments are consed to the last argument:

```
            (apply map + a) -> (map + a)
(apply list 2 3 4 '(5 6 7)) -> (list 2 3 4 5 6 7)
```

`Map` is a variadic procedure which accepts a number of lists, so the structure it ultimately processes is a list of lists. Like *mapcar* it passes the car parts of these lists to its procedure argument. It then advances to the cdr parts of its list arguments.

The version of `map` that will be constructed here will stop as soon as it reaches the end of one of the lists passed to it. Subsequently it processes lists only up to the length of its shortest argument:

```
(map + '(1 2 3) '(4 5)) => (5 7)
```

Note that this behavior, although found in some environments, does not comply with the Scheme standard, which states that all lists passed to `map`

must have the same length.

A procedure for testing whether one member of a list is equal to `()` can easily be created by first introducing another useful predicate: *any?* tests whether a given list *a* contains any member satisfying a predicate *p*:

```
(define (any? p a)
  (cond ((null? a) #f)
        ((p (car a)) #t)
        (else (any? p (cdr a)))))
```

Using this predicate, the *any-null?* procedure is trivial:

```
(define (any-null? x) (any? null? x))
```

Procedures for taking the car parts and the cdr parts of the sublists of a list can be defined by means of *mapcar*:

```
(define (car-of a) (mapcar a car))
(define (cdr-of a) (mapcar a cdr))
```

Given these procedures, the definition of `map` is quite straight-forward. The procedure basically looks like *mapcar*, but uses *car-of* instead of `car`, *cdr-of* instead of `cdr` and *any-null?* instead of `null?`. And because a real `map` has to run in constant space, the following version is tail-recursive, too:

```
(define (map f . a)
  (letrec
    ((map2
       (lambda (a b)
         (cond ((any-null? a)
                 (reverse b))
               (else
                 (map2 (cdr-of a)
                       (cons (apply f (car-of a)) b)))))))
    (map2 a '())))
```

Note that this procedure would accept zero lists, which the real `map` procedure does not. However, as mentioned earlier, standard Scheme provides no procedure for indicating failure, so this condition is left unchecked. (You may use `bottom` or an implementation dependent error reporting facility to improve this version of `map`.)

One question that comes into mind is this: If `map` can do anything *mapcar* can do, can the occurrences of *mapcar* in *car-of* and *cdr-of* be replaced with `map`? Can *mapcar* be optimized away completely by doing this? The startling answer is: "no". (Before you go ahead: can you figure out on your own why this optimization is not possible?)

The answer is given in a semi-formal way by assuming that *car-of* is written in this way:

```
(define (cdr-of a) (map cdr a))
```

Given this definition, a sample application of `map` is reduced:

```
(map - '(1 2 3))
-> (map2 ((1 2 3)) ())
-> (cond ((any-null? ((1 2 3))) (reverse b)) ...)
-> (cond (else (map2 (cdr-of ((1 2 3))) ...)))
```

The ellipses in the above reduction denote that this part of the expression is currently of no interest. What *is* interesting at this point is that

```
(map cdr ((1 2 3)))
```

has to be evaluated in order to get the value of first argument of *map2*. It is evaluated like this:

```
(map cdr ((1 2 3)))
-> (map2 ((1 2 3)) ())
-> (cond ((any-null? ((1 2 3))) (reverse b)) ...)
-> (cond (else (map2 (cdr-of ((1 2 3))) ...)))
```

At this point `(map cdr ((1 2 3)))` has to be evaluated *again* in order to continue the reduction. In other words: "*To compute the first argument of map2, the first argument of map2 has to be computed.*" This is a recursive statement without a trivial case, so its reduction can never terminate.

No reduction containing the same intermediate step twice (like above example) can ever reduce to a normal form, because such a repetition indicates a *cycle* in that reduction. If you can show that the evaluation of an expression contains a cycle, you have proven that the expression in question has no normal form. In terms of programming: you have proven that the program will "crash" when it is run. Do you think that this can be done in imperative languages?

2.3.1 Some Fun with Higher Order Functions

In an earlier section, two versions of the *depth* procedure were shown. The section ended with the question whether *depth* can be written in any simpler way. Using higher order procedures, indeed, it can:

```
(define (depth a)
  (if (pair? a)
      (+ 1 (apply max (map depth a)))
      0))
```

Higher order functions are a quite powerful tool. They actually do add expressiveness to a language, because they allow it to do things that could not be done without them. Another previous section explained why `if` cannot be implemented as an ordinary procedure. The reasoning was as follows. Given a procedure *xif* which implements `if`, the expression

```
(letrec
  ((down
     (lambda (n)
       (xif (zero? n)
            0
            (down (- n 1))))))
  (down 1))
```

would reduce to bottom, because both the trivial and the recursive branch of

```
(xif (zero? n) 0 (down (- n 1)))
```

would be evaluated *before xif* was applied and therefore the procedure could never terminate. Higher order functions help here, too, because they can be used to ‘‘wrap up’’ expressions for later evaluation. Consider the expression

```
(lambda () (down (- n 1)))
```

This expression reduces to a procedure whose body is the branch that causes trouble in the above application of *xif*. Because the troublesome expression is wrapped up in a procedure, it is not evaluated before the procedure is applied (to zero arguments) by writing:

```
((lambda () (down (- n 1))))
```

(Note the additional parentheses!) The following equation holds for any expression *x*:

```
x = ((lambda () x))
```

The following implementation of *xif* is based on this equality:

```
(define (xif pred true false)
  (if (pred) (true) (false)))
```

It accepts three zero-argument procedures which wrap up the predicate, the true branch and the false branch of the conditional expression, so

```
(if a b c)
```

has the same meaning as

```
(xif (lambda () a) (lambda () b) (lambda () c))
```

Therefore above expression involving the application of *down* can be re-written using *xif*:

```
(letrec
  ((down
     (lambda (n)
       (xif (lambda () (zero? n))
            (lambda () 0)
            (lambda () (down (- n 1)))))))
  (down 1))
```

Except for its awkward syntax *xif* is perfectly equal to `if`. The above expression even reduces in constant space, so any value can be passed to *down* safely.

In fact, higher order functions are *so* powerful that `lambda` alone is sufficient to create an entire programming language. Doing so is beyond the scope of this book, but here is a little teaser: In the section explaining primitive procedures, these were described as procedures that cannot be implemented in Scheme itself *easily*. Here is how the basic list processing primitives can be re-written in Scheme using only `lambda` and variables:

```
(define (kons a b) (lambda (p) (p a b)))
(define (kar x) (x (lambda (a b) a)))
(define (kdr x) (x (lambda (a b) b)))
```

Let us see if this works:

```
(kons 'head 'tail) => #<procedure (p)>
```

Does not look too promising, does it? But wait. A cons cell is an object that glues together two other objects, where the first one can be extracted using `car` and the second one can be extracted using `cdr`, and this is exactly what *kar* and *kdr* do with objects created using `kons`:

```
(kar (kons 'head 'tail)) => head
(kdr (kons 'head 'tail)) => tail
```

Here is an explanation how it works. Note that the external representations of procedures include bodies here and the variables of procedures are substituted by their values to make the code easier to read.

```
(kons 'head 'tail)
=> #<procedure (p) (p head tail)>
(kdr (kons 'head 'tail))
-> (kdr #<procedure (p) (p head tail)>)
-> (#<procedure (x) (x (lambda (a b) b))>
    #<procedure (p) (p head tail)>)
-> (#<procedure (p) (p head tail)> (lambda (a b) b))
-> ((lambda (a b) b) head tail)
=> tail
```

Even recursion can be implemented using `lambda` exclusively by using *self-application*:

```
(define (S f x) (f f x))
```

Using *S*, the recursive *fact* procedure from the first section can be re-written in this way:

```
(S (lambda (f x)
     (cond ((zero? x) 1)
           (else (* x (f f (- x 1))))))
   5)
=> 120
```

This expression uses *an anonymous recursive procedure* to compute the factorial of 5, but how can a procedure apply itself when it has no name?

A procedure that is passed to *S* has to provide an additional argument which will be bound to a copy of itself. This is what the *f* argument of the above lambda function is used for. (There is a more general device known as the "Y combinator" that does not have this limitation. It will be introduced later.)

S applies the procedure *f* to two arguments: *f* itself and 5. Here is another depiction of the anonymous factorial procedure. This time, *S* is expanded to a lambda function and the first argument of the operator is rendered in boldface characters:

```
((lambda (f x) (f f x))
 (lambda (f x)
   (cond ((zero? x) 1)
         (else (* x (f f (- x 1))))))
 5)
```

You might want to try to evaluate this expression on a piece of paper in order to understand what it does. You may need a lot of paper, though, which is why there is no sample reduction in this book.

2.4 Dynamic Typing

Like most popular languages, Scheme is a *typed* language which means that there are different types of objects, like numbers, booleans, chars, strings, etc. Most contemporary languages like C or Java are statically typed languages, though, while Scheme uses a system called *dynamic typing*. In statically typed languages, types are associated with variables. A variable of a given type can be used to store objects of that type. The dynamic typing approach is much more flexible. It is made possible by the fact that Scheme does not store data in variables but only stores references to data.

Scheme variables do not have types. They are just symbolic names and you can bind any type of object to them. The type information is carried in the objects themselves. For example, the object `127` is of the type *integer*, the object `"Hello, World!"` is of the type *string*, and the object `(a b c)` is of the type *pair*. A value that carries its type information with it is also called a *boxed value*.

Many procedures accept only arguments of one specific type. For instance, the `reverse` procedure expects a list:

```
(reverse '(a b c)) => (c b a)
```

Passing any other type to the procedure will make it fail:

```
(reverse 'non-list) => bottom
```

The actual behavior of the `reverse` procedure depends on the Scheme implementation you are using. Most versions will tell you that you applied a procedure expecting a list to a symbol or print a similarly informative message. In an environment where `reverse` is a user-level procedure, though, the system may not perform any type checking at all when `reverse` is applied to an argument. When the procedure is applied to a non-list, it simply fails as soon as the first primitive is hit. In such an environment the `reverse` procedure may look like this:

```
(define (reverse a)
  (letrec
    ((reverse2
       (lambda (a b)
         (cond ((null? a) b)
               (else (reverse2 (cdr a)
                               (cons (car a) b)))))))
    (reverse2 a '())))
```

When this procedure is applied to the symbol *non-list*, the following happens:

```
(reverse 'non-list)
-> (reverse2 non-list ())
-> (cond ((null? non-list) b)
         (else (reverse2 (cdr non-list)
                         (cons (car non-list) b))))
-> (reverse2 (cdr non-list)
              (cons (car non-list) b))
```

At this point, Scheme attempts to take the cdr of a symbol:

```
(cdr non-list) => bottom
```

and because Scheme procedures are strict, `reverse` also reduces to bottom. The system will then print a message like this:

```
reverse2: cdr: expected a pair, but got: non-list
```

It tells you that the application of `cdr` to an object that is not a pair was attempted. The system also tells you that this happened inside of the `reverse2` procedure and that the offending object was the symbol *non-list*.

It is a common practice in Scheme to let procedures fail like this. Of course, you *could* intercept such type errors by checking the argument

using a predicate like `list?`, but all this would buy you was a slightly improved error message. This is why Scheme programmers normally do not bother with adding such redundant checks.

Note that adding type checks does *not* improve safety, because all primitive procedures of Scheme perform type checking internally. This is why the `cdr` procedure finally caught the bad type of the argument of `reverse`. All primitives abort evaluation immediately when an object of the wrong type is passed to them, and because all procedures are ultimately composed of primitives, a wrong type sooner or later will hit such a primitive. The following example will illustrate this principle a bit more. Let us assume that the `char<=?` predicate was a user-level procedure. It would probably be defined in such a way:

```
(define (char<=? a b) (not (char>? a b)))
```

It is based on the `char>?` predicate, which could be a user-level procedure, too:

```
(define (char>? a b) (char<? b a))
```

`Char<?`, finally, is a primitive procedure which expects two arguments of the type char. Now imagine a procedure called *hex-digit?* which is based upon `char<=?`:

```
(define (hex-digit? x)
  (or (char<=? #\0 x #\9)
      (char<=? #\a x #\f)))
```

When a non-char is passed to *hex-digit?*, evaluation continues until the primititive procedure `char<?` is applied to it. First *Hex-digit?* calls `char<=?`, which calls `char>?`, which in turn calls `char<?`. `Char<?`, finally, detects the wrong type and reduces to *bottom*.

But what happens if a type error is never caught at all? Type errors are feared by programmers all over the world. Once more the answer is: relax. A type error that is not caught in Scheme is not a type error at all. Either an object hits a primitive that does not know how to handle it, or the evaluation terminates normally and delivers a proper normal form.

In fact, there have been quite a few procedures in this book that do not depend on specific types. For example, the `cons` procedure can be used to glue together objects of any type:

```
    (cons 'a #f) => (a . #f)
(cons "foo" 123) => ("foo" . 123)
(cons #\x '(25)) => (#\x 25)
```

The `eq?` procedure also accepts arguments of any type, although it does not always reduce to a valid result. This is not a type error, though, but a deliberate design decision. If the set of types accepted by `eq?` was restricted, `eq?` would not work, as the following examples show:

```
(define (null? x) (eq? x '()))
(define (not x) (eq? x #f))
```

The above definitions of `null?` and `not` fully comply with the Scheme standard (but they are mostly implemented as primitives). Instead of saying that `eq?` allows type errors, it should be considered a procedure that does *both* type checking and checking for identity, because

```
(eq? x y) => #t
```

implies that *x* and *y* have the same type.

There are many other procedures and pseudo functions that accept any type, like `quote` and `pair?`, but one of these procedures is particularly interesting: `equal?`. It is interesting because it makes use of type predicates. A type predicate is used to check whether a given object has a given type. `Pair?`, for instance, returns truth *only* if its argument is a pair. Of course, type predicates have to accept any type of argument themselves, or they could not work. The following type predicates exist in Scheme:

```
boolean?  char?  number?  pair?  port?  procedure?
string?  symbol?  vector?
```

Any data object of the Scheme languages satisfies *exactly one* of the above predicates. No object may have more than one type and all types are different. This principle is called *disjointness of types*. In case you wonder why the `list?` predicate is not in the above list: The list is a special case of the pair, so it is sufficient to include the pair in the list of fundamental types. Furthermore, adding a special case to the list of fundamental types would break disjointness of types because

```
(list? '(a)) => #t
(pair? '(a)) => #t
```

The examples given in this section show that dynamic typing provides

a high level of flexibility and a high level of safety at the same time. It is particularly suitable for rapid prototyping, because it does not force the programmer to make decisions about types where no such decision is required. Statically typed languages that employ strong typing (like Pascal) often lack flexibility, and languages employing weak typing (like C) typically sacrifice safety for flexibility. Dynamic typing combines the safety of strong static typing with the flexibility of weak typing.

Another thing most programmers are concerned about is efficiency, and boxed values decrease performance, because they do not fit in registers. However, optimizing Scheme compilers can decide to ''unbox'' values in contexts where their type information is not needed. For example, when compiling the *distance-between-points* procedure introduced earlier, a compiler may find out that the type information of the arguments is not used:

```
(define (distance-between-points x y x2 y2)
  (let ((dx (abs (- (unbox x2) (unbox x))))
        (dy (abs (- (unbox y2) (unbox y)))))
    (box (sqrt (+ (* dx dx)
                  (* dy dy))))))
```

Therefore it is sufficient to unbox the arguments when the procedure is entered and re-box the result before returning it. Calls to the fictitious procedures *box* and *unbox* have been added to the code above in order to illustrate this principle. An optimizing compiler would add code for boxing and unboxing values on its own. The values of the variables *dx* and *dy* do not have to be boxed at all, because they are local to the scope of the procedure. Hence all intermediate results can be kept in registers inside of *distance-between-points.*

By unboxing values and limiting type checking to primitive procedures, the overhead is reduced to a degree where there is not much of a difference between optimized Scheme code and optimized code generated by a compiler for a statically typed language. On the other hand, this small overhead buys a high degree of freedom and flexibility.

2.5 Type Conversion

Sometimes it would make sense to apply a procedure that is specific to one type to a different type. For example, it would be nice if the `reverse`

procedure could be used to reverse strings. However, `reverse` expects an argument of the type list. This is where *type conversion* comes into play. The `string->list` and `list->string` procedures convert strings to lists and vice versa. With their help, `reverse` can be used to reverse a string:

```
(list->string
  (reverse
    (string->list "Hello, World!")))
=> "!dlroW ,olleH"
```

The conversion of one type to another is *not* the same as ‘‘type casts’’ in weakly typed languages. A type cast tells a compiler that it should treat an object of a given type as an object of another type, even if this is not the case. A type conversion, though, creates a new object of the desired type that resembles the object to be converted. For example, `string->list` creates a list containing the same sequence of characters as a given string

```
(string->list "Hello World!")
=> (#\H #\e #\l #\l #\o #\space #\W #\o #\r #\l #\d #\!)
```

and `list->string` converts a list of characters into a corresponding string.

There are more type conversion procedures in Scheme, like these:

```
char->integer  string->symbol  list->string  list->vector
integer->char  symbol->string  string->list  vector->list
```

There is not a conversion procedure for every possible conversion. For instance, there is no *symbol->list* procedure, because it would rarely be used, and even if you would need it, it could easily be written by composing `string->list` and `symbol->string`:

```
(define (symbol->list x)
  (string->list (symbol->string x)))
```

In early versions of LISP there was no char type, so single characters were represented by single-character symbols, and there was a procedure that decomposed multi-character symbols into lists of single-character symbols. This procedure was called *explode*. Using above definition of *symbol->list*, it can easily be re-written in Scheme:

```
(define (explode x)
  (map (lambda (x)
         (string->symbol (string x)))
       (symbol->list x)))
```

This procedure first converts the symbol passed to it into a list of chars. Then it maps a procedure over that list which converts each member back into a symbol. `String` is a variadic procedure that creates a new string from its arguments:

```
(string #\f #\o #\o) => "foo"
```

The `string->symbol` procedure, which is used to convert a string to a symbol, should be used with care, because it can be (mis)used to create symbols that cannot be accessed by Scheme:

```
(string->symbol ";unreadable") => ;unreadable
```

Given the above definition of *explode*,

```
(explode 'foo) => (f o o)
```

A procedure that composes a new symbol from a list of single-character symbols is a bit harder to implement, because there is a bit of type-checking to do. It has to be done to make sure that the list passed to that procedure really consists of *single-character* symbols. The inverse procedure of *explode* is called *implode*. Here is its code:

```
(define (implode x)
  (letrec
    ((sym->char
       (lambda (x)
         (let ((str (symbol->string x)))
           (if (not (= (string-length str) 1))
               (bottom "bad symbol in implode")
               (string-ref str 0))))))
    (string->symbol
      (list->string (map sym->char x)))))
```

This procedure basically works like *explode*, but in reverse order. It first maps a procedure over *x* which converts the symbols in the list to chars. The resulting list is converted into a string, and the string into a symbol.

Type checking is performed in the local *sym->char* procedure which, as

its name suggests, converts a symbol to a char. It first converts the given symbol into a string, binding the result to *str*. If *str* has not a length of one character, a type error has occurred and the procedure reduces to bottom. If the string has a length of one character, this character is extracted using `string-ref`. (The first character of a string is at position zero, that is why zero is passed to `string-ref`.)

Using this definition of *implode*:

```
(implode '(f o o)) => foo
(implode '(f oo))  => bottom ; bad symbol in implode
```

2.5.1 Arithmetics with Lists

This little digression introduces two type conversion procedures that are not part of Scheme: *int->list* and *list->int*. These two procedures show how bignum arithmetics can be performed in terms of single digits.

The *int->list* procedure converts an integer number into a list of digits:

```
(define (int->list x)
  (letrec
    ((convert
       (lambda (in out)
         (if (zero? in)
             out
             (convert (quotient in 10)
                      (cons (remainder in 10) out))))))
    (if (zero? x) '(0) (convert x '()))))
```

Lists of digits open the way to a simple and portable (although inefficient) method of performing bignum arithmetics. Of course, it is not normally done in this way, but the the approach illustrates the fundamental principles of arithmetics with arbitrary precision. Here is a procedure that adds two lists of digits:

```
(define (add a b)
  (letrec
    ((result
       (lambda (a b c)
         (if (> (+ a b c) 9)
             (- (+ a b c) 10)
             (+ a b c))))
```

```
    (carry
      (lambda (a b c)
        (if (> (+ a b c) 9) 1 0)))
    (add2c
      (lambda (a b r c)
        (cond ((null? a)
            (if (null? b)
                (cons c r)
                (add2c '()
                       (cdr b)
                       (cons (result 0 (car b) c) r)
                       (carry 0 (car b) c))))
          ((null? b)
            (add2c (cdr a)
                   '()
                   (cons (result (car a) 0 c) r)
                   (carry (car a) 0 c)))
          (else (add2c (cdr a)
                       (cdr b)
                       (cons (result (car a) (car b) c) r)
                       (carry (car a) (car b) c)))))))
   (let ((r (add2c (reverse a) (reverse b) '() 0)))
     (if (= (car r) 0) (cdr r) r))))
```

The details of the *add* procedure are not very interesting. It uses the algorithm which most people use for adding numbers on a sheet of paper. Feel free to explore it on your own. The procedure can indeed operate on numbers of any size:

```
(add '(9 9 9 9 9 9 9 9 9 0) '(1 2))
=> (1 0 0 0 0 0 0 0 0 0 2)
```

List->int implements the reverse function of *int->list*. It converts a list of digits to an integer:

```
(define (list->int x)
  (letrec
    ((convert
       (lambda (in out)
         (if (null? in)
             out
             (convert (cdr in)
                      (+ (car in) (* 10 out)))))))
    (convert x 0)))
```

Using *int->list*, *list->int*, and *add*, you can add arbitrarily large numbers without using +:

```
(list->int (add (int->list 1234567890)
                (int->list 9876543210)))
=> 11111111100
```

Of course, *add* is a rather low-level procedure and all it can do is add natural numbers, not even integers. Implementing integer arithmetics and some more interesting procedures performing operations like integer addition, subtraction, multiplication, and exponentation is left as an exercise to the reader. Interestingly, the lowest-level operations are also the most complex ones. Writing a procedure that performs integer exponentation is a rather simple task compared to the implementation of *add*.

2.6 Arithmetics

Because numbers are a very common thing in programs, the first examples in this book used numbers, and many arithmetic procedures already have been introduced in the sections so far. Like other programming languages, Scheme can add, subtract and multiply numbers:

```
(+ 5 7 9) => 21
(- 5 7 9) => -11
(* 5 7 9) => 315
```

Unlike most languages, Scheme allows a variable number of arguments in these procedures. `+` and `*` accept zero or more arguments. They return the neutral element of the corresponding operation when applied to zero arguments:

```
(+) => 0
(*) => 1
```

The `-` procedure expects at least one argument. When a single argument is passed to it, it negates it, and otherwise it computes the difference of its arguments.

You might have observed that none of the programs discussed before the previous section involved a division, though. This is not because Scheme does not have a procedure for dividing numbers, but because there are several of them. The most versatile of them is the `/` function, which takes at least one argument and returns the reciprocal value of a single argument

or the quotient of two or more arguments:

```
(/ 2)      => 1/2
(/ 6 3)    => 2
(/ 6 4)    => 3/2
(/ 6 4 2) => 3/4
```

What is particularly interesting about `/` is that it returns a *rational value*, if its result cannot be expressed as an integer. Rational values are a separate type of number, but most procedures which accept integers accept rationals as well. For example you can pass rational values to `+`:

```
(+ 2/3 1/3) => 1
```

The procedure accepts the rational arguments, adds them, and if the result can be expressed as an integer, it returns one.

The `quotient` procedure limits its result to the integer domain:

```
(quotient 6 3) => 2
(quotient 6 4) => 1
```

`Quotient` implements a function from number theory. Given two numbers *a* and *b*, it computes the largest positive integer *q* so that *b***q*<=*a*. Less formally expressed, this is what you get in most programming languages when you divide integers: the integer part of their quotient. `Quotient` accepts exactly two arguments, which must both be integers. It can never overflow, but when its second argument is larger than its first, it returns zero. When the second argument is zero, its result is undefined:

```
(quotient x 0) => bottom ; deja vu?
```

The `quotient` procedure is accompanied by a bunch of other procedures from number theory. The most frequently used of those are probably `modulo` and `remainder`. Both of them can be used to compute the remainder of an integer division:

```
(remainder 23 5) => 3
(modulo 23 5)    => 3
```

The difference between `modulo` and `remainder` is the sign of their result. When applied to positive values, they evaluate to equal numbers. `(Remainder a b)` is defined as:

```
(- a (* (quotient a b) b))
```

From this formula follows that `remainder` always delivers a result that has the same sign as its first argument (*a*):

```
(remainder +23 +5) => 3
(remainder +23 -5) => 3
(remainder -23 +5) => -3
(remainder -23 -5) => -3
```

`Modulo`, on the other hand, makes use of the following formula to compute the division remainder:

```
(- a (* (floor (/ a b)) b))
```

The `floor` procedure takes an integer, rational, or floating point argument and reduces to the greatest integer that is not greater than that argument. Because `floor` may decrease negative quotients, `modulo` always delivers a result that has the same sign as its *second* argument (*b*):

```
(modulo +23 +5) => 3
(modulo +23 -5) => -2
(modulo -23 +5) => 2
(modulo -23 -5) => -3
```

`Modulo` can be implemented without using `floor`, too. The following method is based on the observation that `floor` only has an effect on the result, if the signs of the operands of `modulo` differ:

```
(floor (/ 23 5))   = (quotient 23 5)   = 4
(floor (/ -23 -5)) = (quotient -23 -5) = 4
(floor (/ 23 -5))  = (floor -4.6)      = -5
(floor (/ -23 5))  = (floor -4.6)      = -5
```

and the remainder is non-zero:

```
if  (remainder x y) => 0  then  (modulo x y) => 0
```

In the cases where `floor` causes this effect, it reduces the quotient by one:

```
(- a (* (- (quotient a b)
           1)
        b))
```

which can be re-written as follows:

```
(- a (* (- (quotient a b)
```

```
            1)
         b))
= (+ (- a (* (quotient a b)
              b))
     b)
= (+ (remainder a b) b)
```

So all `modulo` has to do is to compute the remainder and then add *b* if the remainder is non-zero and the signs of its operands differ. Here is the code (the `negative?` predicate tests whether a number is negative):

```
(define (modulo a b)
  (let ((rem (remainder a b)))
    (cond ((zero? rem) 0)
          ((eq? (negative? a) (negative? b)) rem)
          (else (+ b rem)))))
```

Two interesting procedures which make use of `remainder` are `gcd` and `lcm`, which compute the *greatest common divisor* and *least common multiple* respectively. Like many other arithmetic procedures, they are variadic:

```
       (gcd 51) => 51
(gcd 289 34 51) => 17
       (lcm 17) => 17
 (lcm 16 17 68) => 272
```

There are many other numeric Scheme procedures to discover. A brief and incomplete summary of them is given here:

```
abs             compute the absolute value
even?           test for n mod 2 = 0
odd?            test for n mod 2 =/= 0
denominator     extract denominator of rational
numerator       extract numerator of rational
sqrt            extract square root
expt            exponentiate
complex?        test for type complex
integer?        test for type integer
rational?       test for type rational
real?           test for type real
ceiling         round towards 1
round           round towards closest integer
truncate        truncate real
sin             compute sine
```

Some of these procedures require the implementation of the full *numeric tower*. The full numeric tower encompasses the types ''integer'', ''rational'', ''floating point'', and ''complex'', including representations for inexact numbers.

These concepts are not explained here, because a discussion of the full numeric tower is beyond the scope of this book. You will probably have noticed, though, that there is a *complex* type, and this fact shall not pass unnoticed.

When you take the square root of a negative number in a Scheme environment implementing the full numeric tower, you will get a result of the type complex, as you would expect it in mathematics:

```
(sqrt 2) => 1.4142135623730951
(sqrt -2) => 0+1.4142135623730951i
```

Standard Scheme implementations are not *required* to include the full numeric tower, but most of them do.

2.7 Strings and Characters Recycled

This section will demonstrate some things you can do with strings and characters. In an earlier section, predicates for comparing chars have been introduced, but chars have some other attributes as well. For example, there are upper and lower case characters. Scheme provides a set of predicates for testing the properties of chars such as `char-alphabetic?`, `char-numeric?`, `char-upper-case?`, `char-lower-case?` and `char-whitespace?`. The effects of these procedures are quite obvious, so they will not be explained in detail. The following procedure prints all properties of a given char using these predicates:

```
(define (char-properties x)
  (apply append
    (map (lambda (prop)
           (cond (((car prop) x) (cdr prop))
                 (else '())))
         (list (cons char-alphabetic? '(alphabetic))
               (cons char-numeric?    '(numeric))
               (cons char-upper-case? '(upper-case))
               (cons char-lower-case? '(lower-case))
               (cons char-whitespace? '(whitespace))))))
```

Given this definition,

```
(char-properties #\X) => (alphabetic upper-case)
(char-properties #\0) => (numeric)
(char-properties #\newline) => (whitespace)
```

The *char-properties* procedure is quite straight-forward but rather artifical. It merely serves as a brute-force example for illustrating a large number of char predicates at one time. The only thing that is interesting about it is the fact that it uses another instance of the combination of `apply` and `map`. In this example, `append` is applied to the result of `map` in order to remove the empty lists generated by predicates that where not satisfied. Without applying `append`, the results of *char-properties* would look a bit messy:

```
((alphabetic) () () (lower-case) ())
```

By applying `append`, the sublists of the result get appended. Because appending an empty list to a list *x* yields *x*, the empty lists are eliminated from the result. The creation of flat lists using `apply` and `append` is a useful tool, as will be demonstrated later in this section.

Another thing you can do with chars is to convert their case. The `char-upcase` procedure converts a lower case character to upper case, and the `char-downcase` procedure converts an upper case char to lower case:

```
(map char-upcase   '(#\a #\- #\Z)) => (#\A #\- #\Z)
(map char-downcase '(#\a #\- #\Z)) => (#\a #\- #\z)
```

As you can see, both of these procedures pass through characters that are not to be converted. That is about all you can do with chars in Scheme.

Because strings are more complex than chars, the things you can do with them are much more interesting, too. For instance, you can create a list of all sub-strings of a string or you can create permutations of strings, which is useful for finding anagrams. Here are some sample solutions to these problems.

To extract a sub-string from a string, the `substring` procedure is used. It works as follows:

```
(substring "hello" 1 4) => "ell"
```

The first argument of `substring` is the source string. Its second argument specifies the position of the first character to extract and its third ar-

gument the position of the first character *not* to extract. The position of the first character of the source string is zero (all string procedures of Scheme use this convention).

The third argument of `substring` must not be less than the second argument. The two arguments may be equal, though. In this case, an empty string is extracted:

```
(substring "hello" 3 3) => ""
```

Because of this convention, it is even legal to extract an empty string from an empty string:

```
(substring "" 0 0) => ""
```

Specifying an offset that is outside of the source string or a negative range (second argument greater than third) is undefined:

```
(substring "abc" 3 4) => bottom
(substring "abc" 1 4) => bottom
(substring "abc" 2 1) => bottom
```

Upon success, `substring` creates a *new string* that contains the characters that were extracted from the source string.

Using `substring`, writing a procedure that creates a list of all sub-strings of a string is quite straight-forward. The first step is to create the procedure *fixed-length-substrings* which creates a list of all sub-strings with a fixed length:

```
(define (fixed-length-substrings s n)
  (let ((len (string-length s)))
    (letrec
      ((f-l-subs
         (lambda (pos)
           (if (> (+ pos n) len)
               '()
               (cons (substring s pos (+ pos n))
                     (f-l-subs (+ pos 1)))))))
      (f-l-subs 0))))
```

Using *fixed-length-substrings*, a procedure to extract all possible sub-strings is easy to do. All you have to do is pass the values from 1 to the length of the given string to *fixed-length-substrings* and collect the results.

The `string-length` procedure for computing the length of a string already has been introduced in the code above, so here is the code of the *sub-strings* procedure that returns all sub-strings of a given string:

```
(define (sub-strings s)
  (let ((len (string-length s)))
    (letrec
      ((subs
         (lambda (n)
           (if (> n len)
               '()
               (append (fixed-length-substrings s n)
                       (subs (+ n 1)))))))
      (subs 1))))
```

Here are some sample applications of *sub-strings*:

```
(sub-strings "") => ()
(sub-strings "x") => ("x")
(sub-strings "Scheme")
=> ("S" "c" "h" "e" "m" "e" "Sc" "ch" "he" "em" "me"
"Sch" "che" "hem" "eme" "Sche" "chem" "heme" "Schem"
"cheme" "Scheme")
```

There is no magic involved in any of these procedures. The only interesting part of *fixed-length-substrings* and *sub-strings* is the fact that they use nested applications of `let` and `letrec` in order to save an argument in their inner procedures. For instance, *subs* could have been implemented as a two-argument procedure that carries *len* with it. However, *len* is constant and the creation of a context only has to be done once while the additional argument would have to be passed along each time *subs* recurses. The creation of an additional context is usually cheaper and often more readable than an additional argument.

The second procedure to be constructed in this section is a procedure that generates *permutations* of a string. A permutation of a string is a string of the same length and containing the same characters but in a different order. For example `"bca"` is a permutation of `"abc"`. The *permute* procedure to be written here, however, will produce *all* permutations of a string. For the input `"abc"` it will return

```
("abc" "acb" "bca" "bac" "cab" "cba")
```

The first step towards a solution is, of course, to investigate how permutations are created systematically. Here is a first approach:

For each character C of a string S
– extract C from S, giving a string S' that contains all characters but C;
– attach C to all permutations of S'.

What this solution is lacking is a trival case. In fact there are two, because there are no permutations for empty strings, and there is only one permutation (the string itself) for single-character strings, so here is an improved recipe:

The permutations of a string S are created this way:
– if S is empty, return `()`*;*
– if S contains only one character, return `(S)`*;*
– otherwise:
  for each character C of S:
  – extract C from S, giving the rest S';
  – attach C to all permutations of S'.

The extraction of a single character from a string is done by the `string-ref` procedure which takes a string and a position as arguments and extracts the char at the given position:

```
(string-ref "abcd" 2) => #\c
```

As usual, the first character is at position zero. A procedure that creates a substring containing all characters but the one at a given position is simple, too:

```
(define (all-but-this-char s n)
  (string-append
    (substring s 0 n)
    (substring s (+ n 1) (string-length s))))
```

There is an even simpler solution, though. When a string of the length *n* is *rotated n* times, each of its characters occurs at the first position exactly one time:

```
"abcd"
"bcda"
"cdab"
"dabc"
```

So it would make sense to create all rotations of the input string and then map a procedure over the resulting list that creates *C* and *S'*, computes all permutations of *S'* and then re-attaches *C* to each permutation. But let us do this step by step. The first procedure that is needed is one that creates a rotation of a string. Here it is:

```
(define (rotate s)
  (string-append (rest-of s)
                 (first-of s)))
```

where *rest-of* and *first-of* are defined this way:

```
(define (first-of s) (string (string-ref s 0)))

(define (rest-of s) (substring s 1 (string-length s)))
```

These two procedures have been factored out because they will be useful at a later time, when *C* and *S'* are created. Note that *first-of* extracts a *string* containing the first character and not just a char. This is helpful when appending the output of *first-of* and *rest-of* at a later time.

Rotate rotates a string by appending a string containing its first character to the rest of the string:

```
(rotate "abcd") => "bcda"
```

Based on *rotate*, a procedure for creating *all* rotations of a string is not hard to do. The number of rotations to perform is equal to the length of the string, so the procedure has to iterate `(string-length s)` times to collect all rotations of *s*:

```
(define (rotations s)
  (letrec
    ((rot (lambda (s n)
            (if (zero? n)
                '()
                (cons s (rot (rotate s) (- n 1)))))))
    (rot s (string-length s))))
```

Rotations delivers all rotations of the given string:

```
(rotations "abcd") => ("abcd" "bcda" "cdab" "dabc")
```

All that is left to do now is to write the *permute* procedure itself. The two

trivial cases are easy to implement. The hard part is the procedure to be mapped over the rotations of the input string. It has to extract the first character of each rotation (giving *C* and *S'*), permute the remaining string *S'*, and then attach *C* to each permutation. Because it maps each rotation to a *list of* permutations, it uses `map` to re-attach *C*. This leads to a nested application of `map`, which might look a bit confusing at a first glance, but the procedure will be explained in great detail immediately. Here is the code:

```
(define (permute str)
  (cond ((= (string-length str) 0) '())
        ((= (string-length str) 1) (list str))
        (else (apply append
                     (map (lambda (rotn)
                            (map (lambda (perm)
                                   (string-append
                                     (first-of rotn)
                                     perm))
                                 (permute (rest-of rotn))))
                          (rotations str))))))
```

And here is what happens inside of *permute*: When a string, say `"abc"`, is passed to it, the general case kicks in. It first creates all rotations of the input string:

```
(rotations "abc") => ("abc" "bca" "cab")
```

Then the following procedure is mapped over the list of rotations:

```
(lambda (rotn)
  (map (lambda (perm)
         (string-append (first-of rotn) perm))
       (permute (rest-of rotn))))
```

The first thing this procedure does it to create all permutations of the *rest* of the current rotation. When the recursively called *permute* returns, it has mapped the rest of *one rotation* into a list of its permutations:

```
(permute "bc") => ("bc" "cb")
```

The inner `map` then pastes the string containing the first character of the current rotation to each member of that list, as the following expression demonstrates:

```
(let ((rotn "abc"))
  (map (lambda (perm)
         (string-append (first-of rotn) perm))
       '("bc" "cb")))
=> ("abc" "acb")
```

When the *outer* map finishes, it has created combinations of all possible first/rest pairs:

```
(("abc" "acb") ("bca" "bac") ("cab" "cba"))
```

All that is left to do at this point is to append the sublists, so that a flat list of permutations is returned. This is done by using the combination of `apply` and `append` mentioned earlier in this section.

The *permute* procedure works for strings of any size, but it slows down significantly as the string grows. There is a pretty obvious optimization, though. It is based on the observation that the permutations of each two-character string are equal to the rotations of that string:

```
(permute "ab")   => ("ab" "ba")
(rotations "ab") => ("ab" "ba")
```

The permutations of two-character sub-strings are created quite frequently during evaluations of *permute*. There are *n!* permutations of a string of the length *n*, which means that *permute* recurses

```
(* n (- n 1) ... 2 1)
```

times to create all permutations. To do so, it combines each of the possible *n* first characters with

```
(* (- n 1) (- n 2) ... 2 1)
```

permutations. To do so, it combines

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

possible prefixes with

```
(* (- n 2) (- n 3) ... 2 1)
```

possible permutations. Iterating this formula further finally yields

```
(* n (- n 1) ... 3)
```

prefixes and two permutations. In other words: *n!/2* prefixes are combined

with permutations of two-character strings, so *permute* spends almost *half of its time* generating permutations of two-character strings. Adding another trivial case to handle these strings should speed up the procedure significantly. Here is the optimized code (the additional clause prints in boldface characters):

```
(define (opt-permute str)
  (cond ((= (string-length str) 0) '())
        ((= (string-length str) 1) (list str))
        ((= (string-length str) 2) (rotations str))
        (else (apply append
                     (map (lambda (rotn)
                            (map (lambda (perm)
                                   (string-append
                                     (first-of rotn)
                                     perm))
                                 (opt-permute
                                   (rest-of rotn))))
                          (rotations str))))))
```

Most Scheme environments provide a facility that allows the user to measure some properties of an algorithm. One property that is frequently measured is the number of *conses* during program execution, i.e. the number of memory cells allocated. Using such a facility, the plain version of *permute* can be compared to the optimized version. Here are the results: [3]

```
; Expression            Nodes allocated  Reduction
Steps
(permute "abcde")       3,228,729        574,788
(opt-permute "abcde")   2,374,935        425,690
```

The actual numbers may differ between environments, but these numbers show that the optimized version saves about 35% of the reduction steps and also about 35% of the used memory. This is not too bad for a single-line optimization.

2.7.1 Garbage Collection

Procedures like *permute* seem to allocate *vast* amounts of memory. When

3 Results where obtained using SketchyLISP's `:statistics` option. SketchyLISP is a now obsolete interpreter that was described in the first edition of this book.

creating the permutations of a seven-character string, the procedure allocates about 78 million nodes in the SketchyLISP interpreter. Given a node size of 9 bytes (on a 32-bit architecture), this makes about 702 million bytes, and the procedure does not even seem to contain any code to release unused storage. So how does memory management work in Scheme?

From a programmer's point of view, this is simple: just keep allocating things and Scheme makes sure that unused stuff is recycled and returned to the pool of free memory. Sounds too good to be true? There must be a catch? No, there is none. Automatic memory management has made quite some progress in the past years, and there is really no excuse for managing memory ''manually'' these days. Let us see how automatic memory management works.

Many Scheme procedures allocate memory. In the *permute* procedure, `substring` was used heavily. Each time `substing` is called, it allocates a new string. The same is valid for `string-append`, and even `string-length` might create a boxed integer.

Recursive procedures like *permute* create really huge amounts of intermediate results. `(Opt-permute "abcdefg")` creates about 700M bytes of intermediate data, but delivers just a list of 5040 seven-character strings, allocating about 85K bytes. However, the peak memory load created by *opt-permute* is only about 190K bytes when run in SketchyLISP. [4]

To be able to allocate 700M bytes of memory in a pool that may be as small as one megabyte, Scheme has to recycle unused data on a regular basis. It normally does so when the free space of a memory pool runs low. When this happens, it marks all data that can be accessed at the current time and releases the rest to the pool of free memory (which is also called the *freelist* by LISP programmers).

Note that Scheme actually *proves* that an object is no longer usable before it recycles its memory. It *never* throws away data that you might want to use at a later time. How does this work? Any object can only be accessed as long as it is referred to by a variable or contained in a list (or vector) that is referred to by a variable. When an object is not accessible through any variable any longer, it becomes ''garbage''. For example, when passing

4 You can verify this by using the `:gc` meta command of SketchyLISP.

the string `"Hello, World!"` to `string-length`, this string is bound inside of `string-length` as long as the procedure reduces. As soon as it returns its result, though, the string becomes uninteresting:

```
(string-length "Hello, World!") => 13
```

There is no way to access the string at this point, because it is no longer referred to by any variable. If you want to use a string containing the same letters again, you have to create a new one. The same happens in recursive procedures. This principle will be explained by means of a sample implementation of `string-length`:

```
(define (string-length s)
  (letrec
    ((str-len
       (lambda (x r)
         (if (null? x)
             r
             (str-len (cdr x) (+ 1 r))))))
    (str-len (string->list s) 0)))
```

When a string is passed to this procedure, it is bound to *s*, converted to a list, and then passed on to *str-len* where the list is bound to *x*. At this point there is a string and a list kept in memory (and the integer *r*, but it does not matter in this explanation). When the general case occurs in *str-len*, the cdr part of the list is passed to *str-len*. This causes the cdr part of the original list to be bound to *x*. Because *str-len* is tail-recursive, the original binding of *x* is discarded. At this point, no variable refers to the full list any longer, so the car part of the list becomes garbage. At each iteration, one member of the list becomes useless, and when the trivial case is reached, the entire list is garbage. Once *str-len* is called, the memory usage of the program *decreases*. The program releases memory, and it does so without any explicit instructions to do so.

The process of recycling unused memory is widely known as *garbage collection* (or in short: *GC*). GC got a bad name because many people think that it causes a great overhead and, even worse, may stop the program at random locations in order to reclaim storage. The latter is an artifact from quite early times. Modern garbage collectors do not stop the program to do their task. They work in the background while the program runs. The overhead caused by garbage collection (concurrent or not) is no larger than

the overhead caused by *any* decent memory manager. The only scenario where manual memory management wins is a program that only allocates memory but *never* releases it. As soon as memory has to be released, the two principles run head to head, but GC frees you *completely* from the task of keeping track of used memory. *Using GC, memory leaks cannot occur!* [5]

Another thing that modern GCs avoid is *memory fragmentation*. Memory fragmentation occurs when lots of small regions are randomly allocated and freed over a period of time. Doing so leaves a memory layout like this:

```
XXX...XXX..XXX...XX...XXX...XX..XXX...XX
. = 1 unit of free memory
X = 1 unit of allocated memory
```

Although the above region contains much more than four units of free memory, it is impossible to allocate a four-unit object in it, because the largest contiguous free region has a size of three units. Therefore, a program may report insufficient memory although sufficient memory exists – just not as a contiguous region.

What modern garbage collectors do in such cases is called *memory compaction*. The allocated units are moved to one end of the memory pool and the free units to the other, leading to the following layout:

```
XXXXXXXXXXXXXXXXXXXXX....................
```

After compacting memory, the same pool provides enough space for allocating a 19-unit object.

Automatic memory management does not only deliver the same performance as explicit memory management in non-trivial scenarios, it also makes better use of the existing memory by avoiding fragmentation and it frees the programmer from the tedious and superflous task of keeping track of used memory. Using garbage collection, ‘‘dangling references’’ (references to accidentally freed regions) and memory leaks belong to the past.

5 A program that fails to release unused memory is said to have a memory leak.

2.8 Input, Output, and Side Effects

All Scheme procedures have an *effect* which is to map a number of arguments to a value. This is even true with most pseudo functions. For example `cond` maps a number of clauses to a value. Given the same arguments, a procedure always returns the same value.

There are constructs, though, which have an effect that cannot be explained by mapping arguments to values. In section 1.7, one of them was introduced in order to explain `letrec`: `set!` changes the value of a variable, thereby mutating the state of the current environment. This can easily be observed by checking the value of the variable before and after the application of `set!`:

```
(define x 'foo)
x => foo
(set! x 'bar)
x => bar
```

Because this effect of `set!` on *x* cannot be explained by mapping arguments to values, it is called a *side effect*. There are two common kinds of side effects in Scheme. One is the mutation of objects, and the other is caused by input and output operations.

Scheme, being a multi-paradigm language, provides several constructs for mutating state. The names of these constructs end with an exclamation mark (like `set!`) to indicate that they have side effects on objects.

In this book a ''purely applicative'' subset of Scheme is discussed, so there are only few procedures with side effects. The only constructs discussed here that alter values are `set!` and `letrec`. `Letrec` does this ''behind the scenes'', which is why its name has no trailing exclamation mark.

An *applicative* language is a language in which programs are formed by combining the applicatitions of procedures and pseudo functions. Strictly speaking, it is *purely* applicative, if its procedures do not have any side effects at all. So, strictly speaking again, the language discussed here is not *purely* applicative, because some of its constructs do have side effects.

In fact, terms like ''purely applicative'' or ''purely functional'' are a bit blurry. There are a few real-world languages that claim to be ''pure'' in

this regard, but a real-world language without I/O would not make much sense, so I/O seems to be an exception to the rule.

2.8.1 Input and Output

To cause side effects is in the nature of input and output procedures. In fact many I/O procedures do not even have any observable effect other than returning an unspecific value. The side effects of I/O procedures are often more important than their effects.

The most abstract I/O procedures of Scheme are called `read` and `write`. These procedures translate between internal and external representation. `Read` is used by Scheme environments to parse programs, and `write` is used to write the external representations of normal forms to the user's terminal.

Most output that is written by `write` can be read back using `read`. There are some objects that do not have any external representation, though, like procedures. Such objects are represented by some informative text enclosed by `#<` and `>`:

```
(write cons)                writes  #<primitive cons>
(write (lambda () 'foo))    writes  #<procedure ()>
```

Any attempt to read such a form results in bottom:

```
(read) #<primitive cons> => bottom
```

In case you wonder what applications of `write` reduce to: it is something called an *unspecific value*, frequently represented by `#<unspecific>` or `#<void>`. Unspecific values are used to indicate that the value returned by a procedure or pseudo function is not interesting. You cannot do anything useful with such a value, and they cause a type error when passed to most procedures. Output procedures typically reduce to an unspecific value, because their effect is not important.

`Write` writes its output to the same output stream as the Scheme system itself, so the output of `write` and the system itself are mixed:

```
(write "Hello, World!")
"Hello, World!"=> #<unspecific>
```

You already know `read`, too, because all programs you have typed in so

far were processed by this procedure. In interactive environments, `read` reads the same input stream as the Scheme prompt, so you can place your input right after an application of read

```
(read) (this is a list with members)
=> (this is a list with members)
```

Two interesting things happen above. `Read` is a zero-argument procedure. It evaluates to the object read by it. Because it can evaluate to different values given the same argument (none), it is obvious that it has a side effect. The other interesting part is that the list read by `read` is not quoted. This is not necessary, because the list is never evaluated. Quotation is only needed to tell Scheme that a list is data and not a program. Because something that is read in *obviously* is data, there is no need to quote it. In fact, quoting it would cause `read` to return an application of `quote` instead the quoted object:

```
(read) foo  => foo
(read) 'foo => 'foo
```

Because `read` and `write` translate objects from and to an unambiguous external form, they are useful for storing and retrieving information.

If you want to write some output to a terminal while your program runs, however, the `display` procedure is probably what you want. `Display` "pretty-prints" objects. It does not print quotation marks around strings and emits "special" characters and white space without escaping them. For instance, `display` can be used to begin a new line by emitting a `#\newline` character:

```
(display #\newline)

=> #<unspecific>
```

`Write` would emit the external representation `#\newline` instead. Here are some other output samples of these two procedures:

write	**display**
`#\space`	
`"A \\/ B"`	`A \/ B`
`#\x`	`x`
`"\"Hi!\""`	`"Hi!"`

Because output procedures are called for their side effects, their returned values are normally ignored. The following procedure makes use of this fact. It is used to call the same procedure a given number of times:

```
(define (do-times n f)
  (if (zero? n)
      (void)
      (begin (f)
             (do-times (- n 1) f))))
```

Do-times applies *f* *n* times and then returns a meaningless value using *void*. Note that the *void* procedure does not conform to the Scheme standard, although a lot of implementations do include it. If your Scheme does not provide it, you can define it using:

```
(define (void) (if #f #f))
```

When no alternative is given to `if`, it returns an unspecific value when the (non-existing) alternative is to be evaluated.

What is more interesting about *do-times* is its use of `begin`. This pseudo function is similar to `and` and `or`: it evaluates the expressions passed to it in sequence. Unlike `and` and `or` it does not ever do anything with the normal forms of these expressions, though. It always evaluates all of the given expressions and returns the value of the last one:

```
(begin 'foo) => foo
(begin 'foo 'bar 'baz) => baz
```

Because the values of all but the last expression are discarded, this pseudo function is only interesting if the expressions passed to it have side effects. In *do-times* the *f* procedure is called and its value is discarded. When *f* returns, *do-times* recurses to apply *f* another time. It can be used, for example, to write a string 10 times:

```
(do-times 10 (lambda () (display "Hello")))
HelloHelloHelloHelloHelloHelloHelloHelloHelloHello
=> #<unspecific>
```

In case you want each of the 10 occurrences of "Hello" to be written on a fresh line, `begin` can help, too:

```
(do-times 10 (lambda ()
               (begin
                 (display "Hello")
                 (newline))))
```

`Newline` is a built-in procedure that does the same as `(display #\newline)` but is slightly easier to type.

Note that `begin` is not really necessary in the above expression, because the body of `lambda` contains an *implied* `begin`, which allows you to write:

```
(do-times 10 (lambda ()
               (display "Hello")
               (newline)))
```

The `read` procedure reads only complete forms. Once applied, it waits until an object has been read entirely:

```
(read) (define foo ; this is ignored
(lambda () 'bar)
)
=> (define foo (lambda () 'bar))
```

This is pretty handy when parsing Scheme programs, but makes it hard to read some specific characters alone, such as the semicolon or the opening parenthesis. This is why Scheme has a procedure for reading raw characters, too:

```
(read-char)x => #\x
```

`Read-char` reads a single character and returns it. Because it does not process its input in any way, it can be used to read any character:

```
(read-char)( => #\(
```

Note that `read-char` reads the same input stream as the Scheme system itself, so when you use it at the Scheme prompt, you have to place the character to read *immediately* after the last closing parenthesis. If you leave a blank in between, that blank will be read instead:

```
(read-char) ) => #\space
```

After reading the space in this example, the closing parenthesis will be fed to the Scheme evaluator which will probably complain about it.

There is no built-in procedure for reading lines of text, but such a procedure can easily be constructed using `read-char`.

```
(define (read-line)
  (letrec
    ((collect-chars
       (lambda (c s)
         (cond ((eof-object? c)
                (cond ((null? s) c)
                      (else (apply string (reverse s)))))
               ((char=? c #\newline)
                 (apply string (reverse s)))
               (else (collect-chars (read-char)
                                    (cons c s)))))))
    (collect-chars (read-char) '())))
```

Scheme's input/output procedures are not limited to the screen and the keyboard. They can be used to read and write *files*, too. Scheme uses so-called *ports* to implement access to files, but most port-related procedures work in an imperative way and hence do not integrate well with the procedural paradigm. Only such file I/O procedures that fit well in applicative programs will be discussed here.

The `with-input-from-file` procedure opens an input file for reading:

```
(with-input-from-file "some-file" read-line)
=> "first line of file"
```

The first argument of `with-input-from-file` specifies the file to read. Its second argument must be a procedure of zero arguments. `With-input-from-file` opens the given file and connects the *default input port* to that file. In this context it evaluates the given procedure. When the procedure returns, the default inport port is re-connected to the file or device that was in effect before the call to `with-input-from-file`.

The effect of redirecting the default input port is that all input read via `read` or `read-char` is read from the specified file rather than the user's terminal.

When a file specified in `with-input-from-file` does not exist, the application reduces to bottom:

```
(with-input-from-file "non-existant" read) => bottom
```

You can read more than a single line, character, or form by passing a procedure to `with-input-from-file` that does whatever you want with the input from the given file. The following procedure copies the content of a file to the default output port, effectively typing that file on the screen:

```
(define (type from)
  (with-input-from-file from
    (lambda ()
      (letrec
        ((type-chars
           (lambda (c)
             (cond ((eof-object? c) c)
                   (else (display c)
                         (type-chars (read-char)))))))
        (type-chars (read-char))))))
```

Although the `cond` of *type-chars* has only two clauses, it is not replaced by `if` for a reason. Clauses of `cond`, like the bodies of `lambda`, imply `begin`, so you can place any number of expressions after the predicate of a clause.

The `eof-object?` procedure used in *type* tests whether the object passed to it is the so-called *EOF object*. The EOF object is a unique, unreadable object that is returned by `read` and `read-char` when a read operation is attempted on a file that offers no more input.

`With-input-from-file` has a counterpart named `with-output-to-file`. As its name suggests, it redirects the default output port in the same way as `with-input-from-file` redirects the input port.

The result of specifying an existing file in `with-output-to-file` is *unspecified* by the standard, so one Scheme environment may overwrite the file silently and another one may report an error and abort program execution.

Using `with-output-to-file` and *type*, it is easy to write a procedure that copies the content of one file to another:

```
(define (copy from to)
  (with-output-to-file to
    (lambda ()
      (type from))))
```

Because `(type from)` is reduced in a context where the default output is directed to the file *to*, the content of *from* is typed to that file. `With-input-from-file` and `with-output-to-file` redirect the input and output of all I/O procedures inside of their contexts, [6] so no special preparations have to be made inside of `type`.

6 Unless the I/O procedure specifies an explicit port, but this feature is not discussed here.

3 Some Missing Pieces

3.1 Syntax Transformation

Throughout this book *pseudo functions* were described as something that not really is a function but whose application looks like a procedure application. This is only half of the truth, though. Keywords like `define`, `lambda`, and `quote` form the *syntax* of Scheme. The syntax is the very core of the language. It describes the lexical form of programs. For example, `define` is not really a procedure, but part of the language itself. This is why just typing `define` at the Scheme prompt yields a syntax error in all standard-compliant implementations:

```
define => bottom
```

This happens because the syntax of `define` has only two valid forms which are

```
(define symbol value)
```

and

```
(define (symbol variable ...) body)
```

and just `define` without any parentheses and arguments is neither of them.

What is particularly interesting about Scheme syntax is that you can *extend* it just like you can extend the Scheme procedure library with own procedures. The process that maps user-defined syntax to existing syntax is called *syntax transformation.*

Here is a simple example:

```
(define-syntax when
  (syntax-rules ()
    ((_ predicate . commands)
       (if predicate (begin . commands)))))
```

In this example `define-syntax` is used to create a new pseudo function named *when*. Whenever Scheme finds an application of *when*, it re-writes it according to the `syntax-rules` associated with it. Although other vari-

ants exist, the body of `define-syntax` always consists of an instance of `syntax-rules` in standard Scheme.

`Syntax-rules` provides a set of rules that is used to transform the application of a pseudo function. Each rule consists of two parts, a pattern and a template:

```
(syntax-rules ()
  (pattern1 template1)
  ...
  (patternn templaten))
```

In the above example, the only pattern is

```
(_ predicate . commands)
```

and the associated template is

```
(if predicate (begin . commands))
```

During the transformation, the application of a pseudo function is matched against each supplied *pattern*. The template of the first matching pattern is used to re-write the application.

3.1.1 Pattern Matching

A pattern *P* is matched against a form *F* as follows:

If *P* is a symbol, it matches any form *F*. For example,

```
x matches 123
x matches "hello world"
x matches (foo bar)
```

If *P* is an atom other than a symbol, *F* must be exactly the same atom. For example,

```
123 matches 123
"hello world" matches "hello world"
          #\q matches #\q
```

If *P* is a keyword, *F* must be the same keyword (keywords will be explained below).

If *P* is a pair, *F* must also be a pair and the car and cdr parts of *P* must

match the car and cdr parts of *F*. For example,

```
  (1 . 2) matches (1 . 2)
  (x . 1) matches (foo . 1)        ; x matches anything
  (1 . x) matches (1 . (bar baz))  ; x matches anything
  (1 . x) matches (1 bar baz)      ; same as above!
(1 (2) 3) matches (1 (2) 3)
```

If an *ellipsis* (the symbol `...`) occurs in a list, it matches any number of forms. For example,

```
(x ...) matches (1)
(x ...) matches (1 #\x)
(x ...) matches (1 #\x (foo))
(x ...) matches (1 #\x (foo) "hi")
```

Whenever a symbol of a pattern (including `...`) matches a form, the symbol is bound to that form. For example:

Matching `x` against `(foo bar)` binds `x` to `(foo bar)`.
Matching `(x ...)` against `(1 2 3)` binds `x` to `1` and `...` to `(2 3)`.
Matching `(x . y)` against `(a b c)` binds `x` to `a` and `y` to `(b c)`.

The difference between the patterns `(x . y)` and `(x ...)` will become clear soon.

3.1.2 Substitution

The application of a pseudo function is re-written by replacing it with the template that is associated with the matching pattern. At the same time, each variable of the template is substituted by the value assigned to it while matching the pattern.

Given the user-defined *when* syntax

```
(define-syntax when
  (syntax-rules ()
    ((_ predicate . commands)
       (if predicate (begin . commands)))))
```

and the application

```
(when (= 1 1) (display "yes!") (newline))
```

the pattern of the first (and only) syntax rule

```
(_ predicate . commands)
```

matches the application. While matching the application, it binds

```
_          to  when;
predicate  to  (= 1 1);
commands   to  ((display "yes!") (newline)).
```

The underscore _ is an ordinary symbol that is by convention used to bind the name of the pseudo function itself.

Substituting the symbols `predicate` and `commands` by their values in the template

```
(if predicate (begin . commands))
```

finally yields

```
(if (= 1 1) (begin (display "yes!") (newline)))
```

Note how the dotted pair in the template is used to cons the list of commands to `begin`:

```
(begin . commands)
= (begin . ((display "yes!") (newline)))
= (begin (display "yes!") (newline))
```

The *when* pseudo function is similar to `if`, but instead of an alternative expression it accepts any number of consequent expressions. Because of its implied `begin` it allows you to write

```
(when p x1 x2 x3 ...)
```

instead of

```
(if p (begin x1 x2 x3 ...))
```

Another way to implement the *when* pseudo function is the use of an ellipsis:

```
(define-syntax when
  (syntax-rules ()
    ((_ predicate command ...)
       (if predicate (begin command ...)))))
```

This version works in exactly the same way as the version using an

improper list. However, the ellipsis has another interesting property that distinguishes it from the dotted list. The *fresh* pseudo function makes use of this property:

```
(define-syntax fresh
  (syntax-rules ()
    ((_ (sym ...) expr . exprs)
      (let ((sym (if #f #f)) ...) expr . exprs))))
```

Fresh creates some fresh local variables without assigning any specific values to them. Such a construct could be useful in the implementation of `letrec`, for example. The syntax of *fresh* is similar to the syntax of `let`, and in fact the above syntax transformer re-writes *fresh* using `let`:

```
(fresh (a b c) (list a b c))
```

becomes

```
(let ((a (if #f #f))
      (b (if #f #f))
      (c (if #f #f)))
  (list a b c))
```

Note that a single symbol precedes the ellipsis in the pattern, but the form

```
(sym (if #f #f))
```

precedes the ellipsis in the template. What the syntax transformer does in this case is to replace *each* form matched by the ellipsis with the above template. In

```
(fresh (a b c) (list a b c))
```

the ellipsis matches the symbols `a`, `b`, and `c`. In the template, the ellipsis is replaced with three instances of `(sym (if #f #f))`. In the first instance, *sym* is replaced by `a`, in the second one it is replaced by `b`, and in the last one by `c`.

Here is another example:

```
(define-syntax twins
  (syntax-rules ()
    ((_ x ...)
      (list (quote (x x)) ...))))
```

The effect of the *twins* pseudo functions is as follows:

```
    (twins 1) => ((1 1))
  (twins 1 2) => ((1 1) (2 2))
(twins 1 2 3) => ((1 1) (2 2) (3 3))
```

3.1.3 Recursive Syntax

Recursion works even in syntax definitions. A syntax transformation terminates only if no more opportunities for a syntax transformation can be found in a resulting form. The following pseudo function reverses a list:

```
(define-syntax reverse-syntax
  (syntax-rules ()
    ((_ lst)        (reverse-syntax lst ()))
    ((_ () r)       r)
    ((_ (a . d) r) (reverse-syntax d (a . r)))))
```

When *reverse-syntax* is applied to a list, the application is re-written to an application of *reverse-syntax* to that list and `()` by rule #1: (the `-->` arrow means "*is re-written to*")

```
(reverse-syntax (1 2 3)) --> (reverse-syntax (1 2 3) ())
```

Because the result contains another application of *reverse-syntax*, the transformation continues. Rule #3 is applied three times:

```
(reverse-syntax (1 2 3) ()) --> (reverse-syntax (2 3) (1))
 (reverse-syntax (2 3) (1)) --> (reverse-syntax (3) (2 1))
 (reverse-syntax (3) (2 1)) --> (reverse-syntax () (3 2 1))
```

At this point, rule #2 is invoked which extracts the second argument:

```
(reverse-syntax () (3 2 1)) --> (3 2 1)
```

The resulting list does not contain any applications of syntax transformers, so it is the final result of the transformation. Using *reverse-syntax*, you can reverse a list *before* it is evaluated:

```
(reverse-syntax (7 5 cons)) => (5 . 7)
```

The following recursive syntax transformer implements the `case` pseudo

function, which is part of the Scheme standard:

```
(define-syntax expand-cases
  (syntax-rules (else)
    ((_ key (else expr ...))
      (begin expr ...))
    ((_ key (data expr ...))
      (if (memv key 'data)
          (begin expr ...)
          (if #f #f)))
    ((_ key (data expr . exprs)
            more-cases ...)
      (if (memv key 'data)
          (begin expr . exprs)
          (expand-cases key more-cases ...))))))

(define-syntax case
  (syntax-rules ()
    ((_ key . cases)
      (let ((k key))
        (expand-cases k . cases)))))
```

`Case` is used to select cases based on the value of an expression (which is called the *key* of `case`). Like `cond`, it has one or multiple clauses as arguments:

```
(case key
      ((d1,1 ...) expr1,1 ...)
      ...
      (else       exprn,1 ...))
```

When the key is found in a list of data $(d_{i,1} \ldots)$, the associated expressions are reduced. Like the bodies of `cond`, the bodies of `case` imply `begin`. The `else` clause, which catches any remaining cases, is optional.

The first thing that stands out in the definition of `case` is the *keyword* `else` in `syntax-rules`. Any number of keywords can be defined in the list that forms the first argument of `syntax-rules`. The keywords are local to the syntax transformer being defined.

Rule #1 of the *expand-cases* helper function matches the `else` keyword and re-writes it as follows:

```
(case key (else x1 ...)) --> (begin x1 ...)
```

The second rule makes sure that `case` reduces to an unspecific value when no matching clause is found.

Rule #3 is a recursive rule that re-writes a `case` with multiple clauses to an `if` expression that handles the first clause and whose alternative branch is another application of `case` that handles the remaining clauses.

The `case` definition itself just evaluates the key before passing it to *expand-cases*, thereby avoiding its multiple reduction.

3.2 Quasiquotation

`Quasiquote` is like `quote`, but only *quasi*. At a first glance they look equal:

```
     (quote (1 2 (+ 1 2))) => (1 2 (+ 1 2))
(quasiquote (1 2 (+ 1 2))) => (1 2 (+ 1 2))
```

But `quasiquote` has a companion keyword named `unquote` which, as its name suggests, allows to *unquote* individual parts of a quasiquoted expression:

```
(quasiquote (1 2 (unquote (+ 1 2)))) => (1 2 3)
```

`Quasiquote` and `unquote` have single-character abbreviations (just like `quote`) which are much more handy than their pseudo function forms. The *"`" (backquote)* character equals an application of `quasiquote` and the *"," (comma)* character equals an application of `unquote`:

```
`(1 2 ,(+ 1 2))  =  (quasiquote (1 2 (unquote (+ 1 2))))
```

It is technically possible to quasiquote and unquote single atoms:

```
 `foo  =  'foo
`,foo  =  foo
```

However, quasiquoting an atom is equal to quoting it, and quasiquoting and then unquoting it is equal to not quoting it at all. The major purpose of quasiquotation is the construction of fixed list structures that contain only few variable parts:

```
(let ((var 'x))
  `(lambda (,var)
     (* ,var ,var)))
=> (lambda (x) (* x x))
```

Without using quasiquotation, this expression would have to be written this way:

```
(let ((var 'x))
  (list 'lambda
        (list var)
        (list '* var var)))
```

Not quite as readable, is it? And things get even worse when you start *splicing* lists using `unquote-splicing` (or `,@` in short):

```
                `(1 2 3 ,@(list 4 5 6)) => (1 2 3 4 5 6)
`(1 2 3 (unquote-splicing (list 4 5 6))) => (1 2 3 4 5 6)
```

Like `unquote`, `unquote-splicing` unquotes its argument, but instead of including it in a surrounding list, it splices it into that list:

```
`(1 2 ,@(list 3 4))  =  (append '(1) '(2) (list 3 4))
```

Because `unquote-splicing` splices its argument into a list, it is not valid outside of quasiquoted lists:

```
`,@(list 3 4) => bottom
```

Here is an expression that constructs a form using `unquote-splicing`:

```
(let* ((var 'x)
       (body `((display ,var)
               (newline)
               ,var)))
  `(lambda (,var) ,@body))
=> (lambda (x) (display x) (newline) x)
```

and the same expression without quasiquotation:

```
(let* ((var 'x)
       (body (list (list 'display var)
                   '(newline)
                   var)))
  (append '(lambda)
          (list (list var))
          body))
```

If you see a pattern emerging here, you are not mistaken:

```
          `x = 'x
         `,x = x
        `(x) = (list 'x)
       `(,x) = (list x)
     `(x ,x) = (list 'x x)
      `(,@x) = (append x)
`(x ,x ,@x) = (append (list 'x) (list x) x)
```

We already know that a quasiquoted atom is a quoted atom and a quasiquoted and unquoted atom is an (unquoted) atom. Here are some new observations: 1) A quasiquoted list is an application of `list` to a series of quoted forms. 2) An unquoted form in a quasiquoted list is not quoted. 3) When `,@` occurs in a quasiquoted list, the application of `list` is replaced by an application of `append`, and `list` is applied to each subform except for those that will be spliced.

3.2.1 Metaprogramming

A metaprogram is a program that writes to re-writes a program. *Metaprogramming* is the art of writing metaprograms.

The previous section introduced some simple metaprograms, such as

```
(let ((var 'x))
  `(lambda (,var)
     (* ,var ,var)))
```

which generates the code of a procedure that squares its argument. Here is a more interesting metaprogram that makes heavy use of quotation and quasiquotation:

```
((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))
```

Can you find out what it does without reading ahead?

Programs like the above are widely known as *quines* in computer science, named after the logician W.V.O. Quine. What they do is to print a copy of their own code. Above quine makes use of the fact that Scheme environments print normal forms of expressions. It reduces to its own code which is then printed by the Scheme system. Some systems may print something like

```
((lambda (x) (quasiquote ((unquote x) '(unquote x))))
'(lambda (x) (quasiquote ((unquote x) '(unquote x)))))
```

instead of the original expression, but this is merely its expanded form. When you evaluate it again, you will finally reach a fixed point in the evaluation. Fixed points are an interesting concept in computer science, and we will get back to them in a later section.

In the previous subsection some rules regarding the equivalence of quasiquotation and `list`/`append` were introduced. Using metaprogramming, it should be possible to transform applications of `quasiquote` mechanically into applications of `append` and `list` by applying these rules. Indeed this is possible. The following metaprogram performs this transformation: [7]

```
(define (expand-qq form)
  (letrec
    ((does-splicing?
       (lambda (form)
         (and (pair? form)
              (or (and (pair? (car form))
                       (eq? 'unquote-splicing (caar form)))
                  (does-splicing? (cdr form))))))
     (expand-list
       (lambda (form)
         (if (does-splicing? form)
             (cons 'append
                   (map (lambda (x)
                          (if (and (pair? x)
                                   (eq? 'unquote-splicing
                                        (car x)))
                              (cadr x)
                              (list 'list (expand x))))
                        form))
             (cons 'list (map expand form)))))
     (expand
       (lambda (form)
         (cond ((not (pair? form))
                (list 'quote form))
               ((eq? 'quasiquote (car form))
                (expand (cadr form)))
```

7 *Expand-qq* is not a full quasiquote expander as required by standard Scheme. It does not handle improper lists correctly, ignores vectors, and does not support nested quasiquotation.

```
                ((eq? 'unquote (car form))
                  (cadr form))
                (else (expand-list form))))))
    (expand (cadr form))))
```

The *expand-qq* procedure expands `quasiquote` to `list` and `append` as described in the previous section:

```
       (expand-qq '`(x)) => (list 'x)
      (expand-qq '`(,x)) => (list 'x)
(expand-qq '`(x ,x ,@x)) => (append (list 'x) (list x) x)
```

It also translates the quasiqoted part of the above quine properly:

```
(expand-qq '`(,x ',x)) => (list x (list 'quote x))
```

Verifying that the above result is actually equal to the original quasiquoted form is left as an exercise to the reader.

3.3 Tail-Recursive Programs

Earlier in this book, a tail-recursive program was described as a program that ''does not do anything after calling itself''. However, this is only half of the truth, as even this trivial example illustrates:

```
(define (down n)
  (cond ((zero? n) 0)
        (else (down (- n 1)))))
```

When stating that the application of *down* is the last thing that *down* does in the general case, `cond` was silently excluded. Of course, `cond` is required to distinguish between the general case and the trivial case, so it *has to be* an exception, or tail-recursion could not work at all. It does work because a `cond` with a true predicate can be re-written this way:

```
(cond (#t expr)) = expr
```

As soon as a true predicate is reached, there is nothing left to do for `cond`. This is why bodies of `cond` are safe positions for *tail calls*. A tail call is a call to a procedure that occurs right before the *calling* procedure returns. When a procedure uses tail calls exclusively to implement recursion, it is tail-recursive.

The predicates of `cond` are *not* valid positions for tail calls, because

`cond` has to examine the normal form of each predicate in order to decide whether the corresponding body is to be evaluated. In the following example, the first recursive call to *equ?* is not a tail call (because it is a predicate of `cond`), but the second one is (because it is a body of `cond`):

```
(define (equ? a b)
  (cond ((or (not (pair? a))
             (not (pair? b)))
         (eq? a b))
        ((equ? (car a) (car b))
         (equ? (cdr a) (cdr b)))
        (else #f)))
```

There are other positions that are safe for tail calls. The last argument of `and` is one such position. The last two clauses of the above *equ?* procedure can be replaced by an application of and:

```
(define (equ? a b)
  (cond ((or (not (pair? a))
             (not (pair? b)))
         (eq? a b))
        (else (and (equ? (car a) (car b))
                   (equ? (cdr a) (cdr b))))))
```

The second application of *equ?* is still a tail call. The other arguments of `and` are not safe, because `and` may have to examine *all* of its arguments before it can return.

The same is true with `or` and `begin`: their last arguments are safe, their others are not. In general, all expressions that can be re-written in such a way that only the tail call remains are safe:

```
        (begin (f))  =  (f)
           (or (f))  =  (f)
          (and (f))  =  (f)
    (cond (#t (f)))  =  (f)
(case 'x ((x) (f)))  =  (f)
    (if #t (f) (g))  =  (f)
    (if #f (f) (g))  =  (g)
```

In `if`, *both* the consequent and the alternative are safe, because if the predicate is true, the consequent is safe and the alternative is ignored, and if the predicate is false, the consequent is ignored and the alternative is safe.

The fact that the *last* positions of `and`, `or`, and `begin` are safe can easily be derived from their definitions, as will be demonstrated here using the example of `and`.

`And` evaluates its arguments $a_1..a_n$ in sequence. When one of the arguments $a_1..a_{n-1}$ evaluates to `#f`, the last argument a_n is never evaluated, so it does not matter whether it is a tail call or not. What remains is the case that all arguments up to the last one evaluate to `#t`. In this case,

```
(and a1 a2 ... aN)
= (and a2 ... aN)
= (and aN)
= aN
```

Showing that the last position of `or` is safe works in an analogous way. The last position of `begin` is trivially safe because

```
(begin a1 ... aN)
= (begin aN)
= aN                  ; modulo side effects
```

Note that the fact that only the last expression of `begin` is a tail call position has consequences for all constructs that use an implied `begin`. Only the *last* expression in the bodies of `lambda`, `cond`, `case`, etc is in a tail call position. In the expression

```
(cond (else (f) (g) (h)))
```

the calls to *f* and *g* are no tail calls, only the call to *h* is one.

Although this might not be obvious, the bodies of the binding constructs `let`, `let*`, and `letrec` are also safe. The following version of *down* is tail-recursive:

```
(define (down n)
  (let ((m (- n 1)))
    (cond ((zero? n) 0)
          (else (down m)))))
```

That this procedure is tail-recursive can easily be shown by re-writing `let` to the application of a lambda function:

```
(define (down n)
  ((lambda (m)
     (cond ((zero? n) 0)
```

```
          (else (down m))))
   (- n 1)))
```

The anonymous procedure application is in a tail call position, because it is the last thing to be evaluated in *down*. The recursive call to *down* is also in a tail call position, because it is the last expression in a body of `cond`. Ergo this version of *down* is tail-recursive.

A tail-recursive program may very well consist of multiple procedures which call each other. Recursion involving multiple procedures is called mutual recursion or *indirect recursion*. The following example implements the *even-p* procedure, which checks whether a number is even, in terms of the *odd-p* procedure, which checks whether a number is odd, and vice versa:

```
(define (even-p x)
  (or (zero? x) (odd-p (- x 1))))

(define (odd-p x)
  (if (zero? x) #f (even-p (- x 1))))
```

Of course this definition works only for positive numbers, but this is a circumstantial detail. What is more interesting is the fact that both *even-p* and *odd-p* are tail-recursive. Tail call optimization, which is responsible for turning tail calls into "jumps", is not limited to procedures that call themselves *directly*. Scheme optimizes *all* tail calls no matter whether they are directly recursive, indirectly recursive, or not recursive at all.

3.4 Continuations

Each point in an evaluation has a past and a future. The past is gone and has condensed to a value. The future, which waits for its past to complete, expects exactly that value. For instance, the application of `car` in the following sample expression waits for the completion of the application of `reverse`:

```
(car (reverse '(a b c)))
```

At the point where `reverse` returns, the past of the evaluation has condensed to the value `(c b a)`, and the future can be expressed as a procedure application that awaits that value:

```
(car _)
```

The underscore character represents the value that the future is waiting for.

Note that the procedure waiting for the past is *always* a single-argument procedure, even in expressions like

```
(+ (* 1 2 3) (* 2 3 4))
```

Although each `*` has three arguments and `+` has two arguments, this expression can easily be broken up into a past evaluating to one single value and a future expecting one single argument at any point. The following sample reduction illustrates this:

```
(+ (* 1 2 3) (* 2 3 4))
-> (+ 6 (* 2 3 4))   ; (lambda (x) (+ x (* 2 3 4)))
-> (+ 6 24)          ; (lambda (x) (+ 6 x))
=> 30
```

At the point where the first argument of `+` has been evaluated (to 6), the future of the computation consists of a procedure expecting that value in order to add it to `(* 2 3 4)`. The procedure that is forming the future of the computation is constructed by replacing the _, which marks the position where the past is expected to complete, with a free variable:

```
(+ 6 _)  becomes  (+ 6 x)
```

and then performing the following transformation:

```
(+ x 6)  becomes  (lambda (x) (+ 6 x))
```

This transformation turns the expression into the body of a lambda function which binds the newly introduced free variable. Thereby the expression becomes a procedure.

In fact there were much more opportunities to split the above example expression into a past and a future. Even the sub-expression `(+ 1 2 3)` contains three points where `+` waits for a value:

```
(lambda (x) (+ x 2 3)) ; wait for 1
(lambda (x) (+ 1 x 3)) ; wait for 2
(lambda (x) (+ 1 2 x)) ; wait for 3
```

Each of the numeric arguments evaluates to itself. While it does so, `+` waits for that part of the reduction to complete. At each of these points, the process can be split into a past and a future.

The future of a computation is also called its *continuation*, and the continuation of the current point in the process of an evaluation is called the *current continuation*. For example, the current continuation of `(f (g x))` at the point where `(g x)` returns is `(f _)` or, after transforming it into a procedure:

```
(lambda (x) (f x))
```

Now imagine a procedure that *captures* the current continuation, freezes it, and packages it for later use. Scheme has such a procedure and it is called `call-with-current-continuation`. Because this name is a bit long, though, many Scheme implementation abbreviate it to `call/cc`. In case your implementation has only the long variant, you can define your own abbreviation using the definition:

```
(define call/cc call-with-current-continuation)
```

`Call/cc` is a single argument procedure expecting a procedure of one argument. What it does is to capture the current continuation and pass it to that procedure:

```
(call/cc f) -> (f #<continuation>)
```

where `#<continuation>` represents the captured continuation.

To prepare the continuation for later use, `call/cc` "reifies" it, which is basically the same as the transformation outlined above:

```
(f _)  becomes  (lambda (x) (f x))
```

So what you get is an ordinary procedure that constitutes the future of a point in the evaluation of an expression. You can "jump" to that future by applying the reified continuation. When you decide to do nothing with a captured continuation, nothing special happens. `Call/cc` returns the return value of the procedure passed to it:

```
(call/cc (lambda (k) 'foo)) => foo
```

When the captured continuation is applied to a value, this value is returned by `call/cc`:

```
(call/cc (lambda (k) (k 'bar))) => bar
```

Of course, the application of *k* does have its own continuation. Because this is not obvious in the above example, here is a better one:

```
(f (call/cc (lambda (k) (g (k 'bar)))))
```

In this example, the continuation captured by `call/cc` is `(f _)`, and the continuation of `(k 'bar)` is `(g _)`.

Applying *k* melts the previously frozen continuation which causes the problem that now there are *two* possible futures for `(k 'bar)`: `(f _)` and `(g _)`. So what Scheme does is to eradicate the current future `(g _)` and establish `(f _)` which is the future captured before:

```
(f (call/cc (lambda (k) (g (k 'bar)))))
-> (f (g (#<continuation> 'bar)))
-> (f 'bar)
```

The current continuation of `(k 'bar)` is *never* reached and so *g* is never applied. You could even do total nonsense in the future of the application of a reified continuation – it would not matter as this example illustrates:

```
(call/cc (lambda (k) (#f (k #t)))) => #t
```

The non-sensical application of `#f` is never reached.

The following sections will explain some things you can do with continuations.

3.4.1 Non-Local Exits

The `length` procedure takes a "proper" list as its argument. Checking for an improper list takes linear time, because the complete list has to be traversed before it can be recognized as an improper one:

```
(define (checked-length x)
  (letrec
    ((proper-list?
       (lambda (x)
         (cond ((null? x) #t)
               ((not (pair? x)) #f)
               (else (proper-list? (cdr x))))))
     (length
       (lambda (x)
         (cond ((null? x) 0)
               (else (+ 1 (length (cdr x))))))))
    (and (proper-list? x) (length x))))
```

The *checked-length* procedure returns the length of a list or `#f` in case an improper list is passed to it. To do so, it has to traverse the list twice: once to check whether the last cdr part of the list is `()` and another time to compute the length. Of course, it would be nice if both of these tasks could be performed in one loop. Unfortunately, this is not that simple:

```
(define (broken-length x)
  (cond ((null? x) 0)
        ((not (pair? x)) #f)
        (else (+ 1 (broken-length (cdr x))))))
```

While this procedure works fine for lists and correctly detects atomic arguments, its application to a dotted list has no normal form:

```
(broken-length '(a . b))
-> (+ 1 (broken-length 'b))
-> (+ 1 #f)
=> bottom
```

At this point, a continuation captured by `call/cc` could avoid the application of `+` to `#f`. The following version of *checked-length* uses `call/cc` to implement a so-called *non-local exit*:

```
(define (checked-length-nl x)
  (call/cc (lambda (exit)
    (letrec
      ((length
         (lambda (x)
           (cond ((null? x) 0)
                 ((not (pair? x)) (exit #f))
                 (else (+ 1 (length (cdr x))))))))
      (length x)))))
```

A non-local exit is a facility that bypasses the normal flow of evaluation. The *broken-length* procedure uses a local exit by just returning `#f` when an improper list is detected. However, this value might be passed back to "outer" instances of *broken-length* which attempt to apply `+` to this value. In the *checked-length-nl* procedure, the *exit* continuation is used to exit from *all* recursively called instances of the *length* procedure *at once*. This is how it works:

The first thing that is applied inside of *checked-length-nl* is `call/cc`. It captures the continuation of its own application and passes it to its argu-

ment which binds the continuation to *exit*. Inside of this context, *length* is defined and applied to the list. When *length* detects an improper list, it applies *exit*. This application discards the current continuation of `(exit #f)` and establishes the continuation saved at the beginning:

```
(checked-length-nl '(a . b))
; the exit continuation is captured here
-> (length (a . b))
-> (+ 1 (length b))
-> (+ 1 (exit #f))
=> #f
```

You might wonder what the continuation captured by `call/cc` in *checked-length-nl* looks like, because there does not seem to be anything to do after the return of `call/cc`. In fact there *is* something to to: *checked-length-nl* has to return the value passed to the continuation. This operation can be thought of as applying the *identity function* `(lambda (x) x)`:

```
(+ 1 (length b))
-> (+ 1 (exit #f))
-> ((lambda (x) x) #f)
=> #f
```

Because adding an application of the identity function to any expression does not change the meaning of that expression, this is a useful tool for understanding continuations in expressions like this:

```
(call/cc (lambda (k) (#f (k 'foo))))
```

What does the continuation that is bound to *k* in this expression look like? Adding an application of the identity functions clarifies things:

```
((lambda (x) x) (call/cc (lambda (k) (#f (k 'foo)))))
```

The continuation is an application of identity:

```
((lambda (x) x) _)
```

3.4.2 Exposing some Gory Details

What makes continuations special is that they are *first class objects* with *indefinite extent*. A first class object is an object that can be bound to variables, passed to procedures and returned by procedures. You can do this with all Scheme objects except for keywords. An object with indefinite ex-

tent exists as long as at least one variable refers to it (directly or indirectly through a list or vector). Only when the last reference to the object is broken, the object becomes garbage and is recycled. All Scheme objects have indefinite extent.

Because continuations are first class objects *and* have indefinite extent, they can escape from the *dynamic context* of `call/cc`, as the following trivial example illustrates:

```
(call/cc (lambda (k) k)) => #<continuation>
```

The dynamic context of `call/cc` is the procedure passed to it. When this procedure returns, the dynamic context of `call/cc` vanishes. In the above example, the identity function returns the continuation passed to it, so when the application of `call/cc` returns, the continuation can be bound to a symbol for later use. Even resuming the continuation at a later time does not invalidate it. Because it captures the *complete future* of a point in an evaluation, it can be applied *any number of times*, which opens the door to a whole load of brain-twisting puzzles. Here is a rather simple one:

```
(let ((x (call/cc (lambda (k)
                    (cons k 'foo)))))
  (let ((k (car x))
        (v (cdr x)))
    (cond ((eq? v 'foo) (k (cons k 'bar)))
          ((eq? v 'bar) (k (cons k 'baz)))
          (else v))))
```

The value *x* is bound to consists of a continuation and a symbol:

```
(#<continuation> . foo)
```

Because continuations are first class objects, they can be stored in structures like lists or pairs. The second `let` decomposes the structure, binding the continuation to *k* and the symbol to *v*. The next action depends on the symbol. Because it is `foo` when `cond` is entered, the following body is evaluated:

```
(k (cons k 'bar))
```

And this is where the fun begins. The object that is passed to the continuation is equal to the original structure only with `foo` replaced by `bar`:

```
(#<continuation> . bar)
```

The continuation stays the same. But what does the captured future look like in this example? It is the point where the reduction of a value to be bound in `let` returns, but the corresponding binding is not yet established. When *k* is applied, the flow of the program kind of *travels back in time* to the point where the value returned by `call/cc` is bound to *x*. Instead of the originally specified value

```
(#<continuation> . foo)
```

`call/cc` this time returns

```
(#<continuation> . bar)
```

To `let` there is no difference to the first time the value of *x* reduced. It binds the new value and evaluates the `cond` expression in its body. The same procedure is carried out again, this time replacing `bar` with `baz`. The case that *v* is bound to `baz` is not specified in `cond`, so it falls through and the value of *v* is returned. `Let` returns `baz` and the continuation is finally recycled.

And now for the gory details that this section promised to reveal. In fact `call/cc` is not hard to grasp. What *is* hard to grasp are all the subtle little details that the Scheme language encompasses. `Call/cc` discloses some details that would pass unnoticed without its existance. Whether this is a good thing or not is debatable.

The order of evaluation of arguments is undefined in Scheme. This means that the expressions passed to a procedure expecting multiple arguments can be evaluated in any order. You cannot tell which of the arguments *a*, *b*, and *c* in

```
(f a b c)
```

will be evaluated first, which second, and which last. Any combination is possible. As long as the sub-expressions *a*, *b*, and *c* do not have any side effects, the order of evaluation does not change the meaning of the application, so each Scheme implementation is free to choose the order of evaluation which is most efficient. If multiple sub-expressions have side effects, these side effects occur in some unspecific order:

```
(letrec
  ((print-and-return
     (lambda (x)
       (display x)
       x)))
  (+ (print-and-return 2)
     (print-and-return 3)
     (print-and-return 4)))
=> 9
```

While this expression is guaranteed to evaluate to 9, it may print any permutation of the values 1, 2, and 3. This is the reason why `begin` cannot be expressed as a lambda function:

```
(lambda x (car (reverse x)))
```

would evaluate its arguments and return the normal form of the last one of them, but the order in which the arguments are evaluated would be undefined.

The following rather simple example exposes the order of sub-expression evaluation without using I/O:

```
(call/cc (lambda (k)
           (#f (k 'left-to-right)
               (k 'right-to-left))))
```

Because evaluating either of the both applications of *k* abandons the rest of the evaluation of this expression and returns the argument of *k* immediately, the expression will reduce to the argument of the *k* that is applied first. However, the result obtained from this expression does not really help. All it demonstrates is that the order of evaluation is left-ro-right or right-to-left in this particular situation. It does not consider the fact that an optimizing compiler still might re-order arguments of other applications in any way that may lead to more efficient code.

The following expression shows that `letrec` actually does mutate bindings in order to create a recursive structure: [8]

8 This test was posted to Usenet by Al Petrofsky on 2005-03-23 using Message-ID <87ll8effvu.fsf@petrofsky.org>.

```
(letrec ((x (call/cc list)))
  (if (pair? x)
      ((car x) (lambda () x))
      (pair? (x))))
=> #f
```

As in an earlier example, *x* is bound to a structure (this time a list) containing a continuation. A list is a pair, so the continuation is applied to

```
(lambda () x)
```

`Lambda` freezes the outer context which includes the current binding of *x*. The value of *x* is still a list containing a continuation at this point. The binding is included in the lexical environment of the resulting procedure. Lexical environments of procedures cannot be changed by applicative programs like the above one, so the value of *x* should not change.

The procedure is passed as an argument to the continuation `(car x)`. Evaluation continues at the point where `letrec` binds its arguments. This time *x* is bound to the procedure returned by `(lambda () x)`.

A procedure is not a pair, so

```
(pair? (x))
```

is evaluated by `cond` and obviously

```
(x)
```

evaluates to a non-pair, or the whole expression would not evaluate to `#f`. However, the value contained in the lexical environment of `(lambda () x)` *was* a pair, so that value must have changed at some point between the creation of the procedure and the application of `cond`.

The only construct that was in between was `letrec`, and what happened is this: When `call/cc` returns for the second time, it evaluates to the procedure resulting from

```
(lambda () x)
```

This procedure contains a lexical environment in which *x* is bound to `(#<continuation>)`. However, *x* is exactly the variable that is bound by `letrec`, so it is a recursive reference. Because `letrec` fixes recursive references, it changes the binding of *x* in the procedure to the value that it

binds to *x*, resulting in the following recursive structure:

```
(lambda () x) where
              x = (lambda () x) where
                                  x = ...
```

To prove that this effect is really caused by `letrec`, it is sufficient to replace `letrec` with `let` and show that the effect does *not* occur in the resulting program:

```
(let ((x (call/cc list)))
  (if (pair? x)
      ((car x) (lambda () x))
      (pair? (x))))
=> #t
```

Indeed this expression evaluates to `#t`, leaving `letrec` as the only possible cause for the mutation of the lexical environment.

3.5 Lambda Calculus and the Y Combinator

Earlier in this book, the self-application combinator *S* was introduced as a device for the construction of anonymous recursive functions:

```
(lambda (f x) (f f x))
```

A *combinator* is a function that does not make use of any free variables, which is why it cannot even use basic operations like `+` or `car`. So it is basically limited to binding constructs, function applications, and its (bound) variables.

While the self-application combinator works fine, it has one limitation. It requires two arguments: the function to be applied and the value the function is to be applied to. It would be much more elegant, if it could just *transform* a function into a recursive function. Such an operator would turn the following factorial function (which does not work because *(lambda (x) ...)* closes over the free variable *f*) into a working recursive function *R*:

```
(lambda (f)
  (lambda (x)                        ----------------->  R
    (cond ((zero? x) 1)                  mystery
          (else (* f (- x 1))))))    transformation
```

Indeed a function performing this ''mystery transformation'' exists. It is called the *Y combinator* (*Y*), and it turns any function *f* of the above form into a recursive function *f'*:

```
(Y f) => f'
```

F' is also called the *fixed point* of *f*, which is why *Y* is also known as the ''fixed point operator''. The fixed point of a higher order function *f* is a function *f'* which is mapped to itself by *f*:

```
(f f') => f'
```

Because the Y combinator computes the fixed point of a function *f*,

```
(Y f)  =  (f (Y f))  =  (f (f (Y f))) ...
```

In the remainder of this section, the definition of a recursive function using `letrec` will be transformed into an equivalent definition that employs the Y combinator instead of `letrec`. A definition of the Y combinator itself will be a by-product of this transformation.

Note that most of the steps (exceptions: 11,12,13,14) during the transformation still implement the factorial function. You can try them in a Scheme environment, if you want to.

Here is the expression to be transformed. It contains a definition of a recursive factorial function using `letrec`:

```
; The original factorial function f
(letrec
  ((f (lambda (a)
        (cond ((zero? a) 1)
              (else (* a (f (- a 1))))))))
  (f 5))
=> 120
```

The use of `letrec` can easily be avoided by adding an argument to *f* that is used to carry a copy of the function itself. It is no coincidence that this argument is also called *f*; it names the function locally. Applications of *f* have to pass along the additional argument, so *(f x)* becomes *(f f x)*:

```
; Step 1: replace LETREC with self-application
(let ((f (lambda (f a)
           (cond ((zero? a) 1)
                 (else (* a (f f (- a 1))))))))
  (f f 5))
```

`Let` can be transformed into an application of a lambda function, which is done in the following step:

```
; Step 2: transform LET to LAMBDA
((lambda (f) (f f 5))
 (lambda (f a)
   (cond ((zero? a) 1)
         (else (* a (f f (- a 1)))))))
```

The above version applies the constant 5 to the factorial function. Because a real factorial function would be applicable to any positive value, though, an argument is added to receive that value:

```
; Step 3: make argument a variable
((lambda (f x) (f f x))
 (lambda (f a)
   (cond ((zero? a) 1)
         (else (* a (f f (- a 1))))))
 5)
```

The above step results in a two-argument factorial function that expects a copy of itself as well as the value whose factorial is to be computed. To convert the two-argument function into a single-argument function, a method known as *currying* is applied.

By currying a two-argument function, that function is turned into a higher-order single-argument function. Applying a curried two-argument function to the first argument and then applying the resulting function to the second argument is equivalent to applying the original function to two arguments. Here is an example. Currying the function

```
(define (add a b) (+ a b))
```

gives

```
(define (add-n a) (lambda (b) (+ a b)))
```

Applying *add-n* to a value evaluates to a single-argument function that adds the given value to its argument:

```
(add-n 5)
; This procedure will add 5 to its argument:
=> #<procedure (b)>
```

Applying this function to a second argument yields the same result as the

application of the original function to the same values at once:

```
(add 5 7) => 12
((add-n 5) 7) => 12
```

Currying the above version of the factorial function separates the function *f* from its argument *a*, just like the above example separated the first argument (5) from the second argument (7):

```
; Step 4: curry the lambda functions
(((lambda (f) (lambda (x) ((f f) x)))
  (lambda (f)
    (lambda (a)
      (cond ((zero? a) 1)
            (else (* a ((f f) (- a 1))))))))
 5)
```

The factorial function *(lambda (f a) ...)* turned into *(lambda (f) (lambda (a) ...))*, self application turned into *(lambda (f) (lambda (x) ...))*, and applications of the form *(f f x)* turned into *((f f) x)*. When *(lambda (f) (lambda (a) ...))* is applied to itself, it closes over its own copy, giving a single-argument function that is capable of applying itself.

The function resulting from this transformation can now be separated from its argument:

```
; Step 5: extract the function
((lambda (f) (lambda (x) ((f f) x)))
 (lambda (f)
   (lambda (a)
     (cond ((zero? a) 1)
           (else (* a ((f f) (- a 1))))))))
```

This expression looks fine, but it has one minor flaw: *f* has to apply itself each time it wants to recurse. The remaining transformations will draw this self-application into the Y combinator itself, so that recursive applications of the factorial function will be ordinary function applications.

The first step to achieve this goal is to give self-application a name (*s*):

```
; Step 6: give self-application a name
((lambda (f) (lambda (x) ((f f) x)))
 (lambda (f)
   (let ((s (lambda (x) ((f f) x))))
```

```
      (lambda (a)
        (cond ((zero? a) 1)
              (else (* a (s (- a 1)))))))))
```

After naming self-application *s*, the recursive function invocation looks like an ordinary function application. The `let` that is used to name self-application is now transformed into the application of a lambda function:

```
; Step 7: transform LET to LAMBDA
((lambda (f) (lambda (x) ((f f) x)))
 (lambda (f)
   ((lambda (s)
      (lambda (a)
        (cond ((zero? a) 1)
              (else (* a (s (- a 1)))))))
    (lambda (x) ((f f) x)))))
```

The next step is to give the factorial function a name. This might look a bit counter-intuitive at a first glance, because this whole exercise is about the creation of *anonymous* recursive functions, but it is a necessary intermediate step. The factorial function is named *g*:

```
; Step 8: give the factorial function a name
((lambda (f) (lambda (x) ((f f) x)))
 (lambda (f)
   (let ((g (lambda (s)
              (lambda (a)
                (cond ((zero? a) 1)
                      (else (* a (s (- a 1)))))))))
     (g (lambda (x) ((f f) x))))))
```

In the next step, `let` is once more transformed into `lambda`, giving an expression that may cause serious headache when examining it. Here are some hints: The *(lambda (s) ...)* part still introduces the factorial function, and the *(lambda (g) ...)* part binds that function to *g* and applies it to the function implementing self-application. Self-application is thereby bound to *s*.

```
; Step 9: transform LET to LAMBDA
((lambda (f) (lambda (x) ((f f) x)))
 (lambda (f)
   ((lambda (g)
      (g (lambda (x) ((f f) x))))
    (lambda (s)
```

```
      (lambda (a)
        (cond ((zero? a) 1)
              (else (* a (s (- a 1)))))))))))
```

In fact, the above expression comes pretty close to a definition of the Y combinator. It only has to be isolated. This is done by ''eta conversion''. The parts of the expression that are added by this conversion are emphasized using boldface characters in the following step.

```
; Step 10: eta-expand Y
((lambda (r)
   ((lambda (f) (lambda (x) ((f f) x)))
    (lambda (f) ((lambda (g)
                   (g (lambda (x) ((f f) x))))
                 r))))
 (lambda (s)
   (lambda (a)
     (cond ((zero? a) 1)
           (else (* a (s (- a 1)))))))))
```

Eta conversion works in two directions. It converts a function into an equivalent lambda function and vice versa:

```
(lambda (x) (f x))  <-------> f
                       eta
```

The transformation of a lambda function to an equivalent function is called eta reduction, the reverse operation is called eta expansion. The conversion performed in step 10 is an eta expansion.

After performing this conversion, the expression can be decomposed into two parts: the factorial function and the Y combinator. Removing the outermost application and the factorial function yields the Y combinator:

```
; Step 11: extract Y
(lambda (r)
  ((lambda (f) (lambda (x) ((f f) x)))
   (lambda (f) ((lambda (g)
                  (g (lambda (x) ((f f) x))))
                r))))
```

Each expression of the form *((lambda (x) t) v)* can be reduced to *t'*, where *t'* equals *t* with each free occurrence of *x* replaced by *v*:

```
((lambda (x) (x x)) v) --------> (v v)
                          beta
```

This conversion is called *beta reduction*. It is exactly this conversion which substitutes arguments for variables when applying a Scheme function. In the next step, the application of *(lambda (g) ...)* is beta-reduced by substituting *r* for *g* in *(g (lambda (x) ((ff) x)))*:

```
; Step 12: remove lambda(g) by beta reduction
(lambda (r)
  ((lambda (f) (lambda (x) ((f f) x)))
   (lambda (f) (r (lambda (x) ((f f) x))))))
```

Finally, the first self-application in the expression is simplified by eta reduction. (In case you wonder why the second self-application is not reduced: this will be explained in the following sub-section.) The result of this step is a definition of the Y combinator:

```
; Step 13: remove first lambda(x) by eta-reduction
(lambda (r)
  ((lambda (f) (f f))
   (lambda (f) (r (lambda (x) ((f f) x))))))
```

To facilitate further experiments with this combinator, it will be named *Y*:

```
; Step 14: name the combinator
(define (Y r)
  ((lambda (f) (f f))
   (lambda (f) (r (lambda (x) ((f f) x))))))
```

The following definition binds *f* to the factorial function of step 8. As you can see, it employs neither `letrec` nor self-application:

```
(define (f s)
  (lambda (a)
    (cond ((zero? a) 1)
          (else (* a (s (- a 1)))))))
```

Given these definitions of *Y* and *f*:

```
(Y f) => #<procedure (a)>
((Y f) 5) => 120
((f (Y f)) 5) => 120
((f (f (Y f))) 5) => 120
...
```

So

– *Y* turns *f* into a recursive factorial function;
– *(Y f)* is the fixed point of *f*.

3.5.1 Scheme vs Lambda Calculus

Lambda calculus (LC) is the basis of all programming languages of the LISP family. Above conversion rules, like eta conversion or beta reduction, actually belong to lambda calculus, but many of these rules apply to Scheme as well. In a way, Scheme may be considered a kind of ''applied lambda calculus''. This is why the Y combinator can be implemented in both LC and Scheme.

This sub-section will explain how the Y combinator works and why the second self-application of the combinator may not be eta-reduced in Scheme.

In fact, the operation performed by *Y* is rather simple: ''*Y* is a function of a function *f* that applies self-application to the application of *f* to self-application.''

Although this description is brief and correct, an example may be more helpful. The following reduction shows how the anonymous factorial function (which already has been stretched a bit far at this point) is converted into a recursive function:

```
(lambda (r)
  ((lambda (f) (f f))
     (lambda (f) (r (lambda (x) ((f f) x))))))
 (lambda (s)
   (lambda (a)
     (cond ((zero? a) 1)
           (else (* a (s (- a 1))))))))
```

In the first step, the factorial function [*(lambda (s) ...)*] replaces the variable *r* in the body of the combinator by beta reduction:

```
; beta-reduce ((lambda (r) ...) ...)
-> ((lambda (f) (f f))
      (lambda (f) ((lambda (s)
                     (lambda (a)
                       (cond ((zero? a) 1)
                             (else (* a (s (- a 1)))))))
                   (lambda (x) ((f f) x)))))
```

In the following step, the factorial function is applied to self-application. This step makes *(lambda (a) ...)* close over *s*, giving a function that is capable of performing self-application. For improved readability, *(lambda (x) ((f f) x))* is eta-reduced to *(f f)* in the same step:

```
; beta-reduce ((lambda (s) ...) ...)
; eta-reduce (lambda (x) ((f f) x))
-> ((lambda (f) (f f))
    (lambda (f)
      (lambda (a)
        (cond ((zero? a) 1)
              (else (* a ((f f) (- a 1)))))))))
```

Finally, the resulting function is applied to itself. It thereby closes over its own copy, leaving a function that applies itself to itself. This transformation is a bit tricky, because *f* is bound in different contexts. To avoid this ambiguity when performing beta reduction manually, lambda calculus provides a rule known as alpha conversion.

Alpha conversion replaces the names of variables that are bound in a function with names that are not bound in the argument of that function. More formally: given the expression

```
((lambda (x) t) v)
```

it replaces all occurences of *x* in *(lambda (x) t)* with a name that does not occur in *v*. This transformation is safe because changing the name of a bound variable does not change the meaning of an expression:

```
(lambda (f) (f f))  =  (lambda (x) (x x))
```

Scheme environments do not have to do alpha conversion because computers are good at keeping track of different instances of the same symbol, but to humans dealing with expressions like the above, it is a valuable aid:

```
; alpha-convert (lambda (f) (f f))
-> ((lambda (x) (x x))
    (lambda (f)
      (lambda (a)
        (cond ((zero? a) 1)
              (else (* a ((f f) (- a 1)))))))))
```

Above self-application makes the function *(lambda (a) ...)* close over *f*:

```
; beta-reduce ((lambda (x) (x x)) ...)
=> (lambda (a)
     (cond ((zero? a) 1)
           (else (* a ((f f) (- a 1))))))
   where f = (lambda (f)
               (lambda (a)
                 (cond ((zero? a) 1)
                       (else (* a ((f f) (- a 1))))))))
```

Each time the function chooses to recurse, *(lambda (f) ...)* is applied to itself one more time, thereby creating another instance of the above expression. So the reduction of this expression *may cycle*, effectively implementing recursion.

Although the Y combinator only has been used to create the factorial function in this section, it is of course not limited to this function. Any function of the form

```
(lambda (f) (lambda (x) body))
```

is mapped to its fixed point by *Y*:

```
((Y (lambda (f) (lambda (x) x))) 'foo) => foo
```

The function transformed by *Y* may use *f* to recurse (although *f* is named *length* in this example):

```
((Y (lambda (length)
      (lambda (a)
        (cond ((null? a) 0)
              (else (+ 1 (length (cdr a))))))))
 '(a b c d e))
=> 5
```

A final question remains. Why was the second self-application of *Y* not simplified using eta reduction? This would have lead to the following Y combinator:

```
(define (Ylc r)
  ((lambda (f) (f f))
   (lambda (f) (r (f f)))))
```

Indeed, this is exactly the way the Y combinator is defined in lambda calculus, but it does not work in Scheme, so there has to be a subtle difference between LC and Scheme. The point where the systems differ is their type of evaluation.

While Scheme uses call by value, lambda calculus employs call by name, except it is called *normal order* evaluation there. (And Scheme's evaluation would be called *applicative order* evaluation in LC).

When using the Y combinator of LC in Scheme, indefinite recursion occurs:

```
(Ylc (lambda (f) (lambda (x) x))) => bottom
```

This happens because self-application [*(f f)*] is applied on the spot in the combinator instead of passing it to *r*.

```
(Ylc (lambda (f) (lambda (x) x)))
; do not care about R; it is not reduced anyway
-> ((lambda (f) (f f))
    (lambda (f) (r (f f)))))
-> ((lambda (f) (r (f f)))
    (lambda (f) (r (f f))))
-> (r ((lambda (f) (r (f f)))
       (lambda (f) (r (f f)))))
-> (r (r ((lambda (f) (r (f f)))
          (lambda (f) (r (f f))))))
...
```

When you try *Ylc* in Scheme, the program will finally run out of memory. Eta-expanding the second self-application simulates call-by-name semantics in Scheme and thereby avoids the above indefinite recursion.

The Y combinator demonstrates the full power of higher order functions by implementing generic recursion using `lambda` and function application exclusively. While lambda calculus is a rather theoretical device, Scheme bridges the gap between mathematics and programming. It combines mathematical rigor with the tools that a programmer needs for exploring problems and inventing new algorithms.

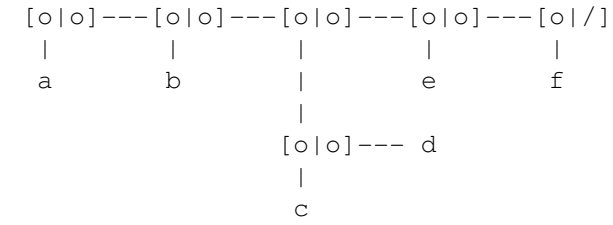
4 Scheme in the Wild

This chapter illustrates what ‘‘real-world’’ Scheme code looks like by means of a small, self-contained, and heavily commented utility program.

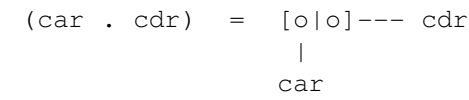
There is a notation called *box notation* which is very useful for understanding Scheme data structures. Using box notation, forms like this one

```
(a b (c . d) e f)
```

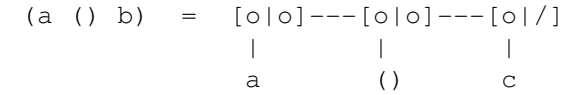
are represented by diagrams like this one:



In a box diagram each object of the type cons is represented by a box: `[o|o]`. The first `o` of the box points to the car part of the cons, the second one to its cdr part:



The empty list `()` is represented by a slash when it occurs at the end of a list:



While drawing box diagrams may be a nice exercise, it is a cumbersome task that begs for automation. The program discussed in the following sections will accept any Scheme datum as input and write the corresponding box diagram to its output.

4.1 Drawing Box Diagrams

The *draw-tree* program, whose code will be shown in the following

section, draws ASCII box diagrams that represent Scheme data structures, just like the diagrams introduced in the previous section. In fact *draw-tree* was used to render these diagrams.

When designing the program, the initial question is, of course, how to draw the diagrams.

So what does a typical Scheme data structure look like? Here is an example:

```
(a (b c . d) e)  =  [o|o]---[o|o]---[o|/]
                      |       |       |
                      a       |       e
                              |
                            [o|o]---[o|o]--- d
                              |       |
                              b       c
```

The "*spine*" of the structure, which is to be drawn first, is a series of boxes representing the cons cells of the corresponding list. The next line contains bars connecting the spine to the members of that list. The third line consists of the members themselves. Members may be lists, so the process recurses at this point.

One problem is to decide when to render sublists. In the above example, the sublist cannot be drawn directly below the spine, because doing so would allocate the space that is needed to render the `e` member.

The problem is solved by always rendering the rightmost member of a list first. After rendering `e`, we know that there is sufficient space to render the second-to-last member below. Atoms are the exception to this rule. They are always rendered as soon as they are encountered. This is why `a` is displayed immediately in the above example.

The next question to ask is how to represent Scheme data structures internally.

The above approach leads quite directly to the data structure used to store lists internally while rendering them. When a list is encountered, a series of conses is drawn and the list is converted into an internal representation:

```
(a (b c . d) e)   renders   [o|o]---[o|o]---[o|/]
                  gives     ((V) a (b c . d) e)
```

`(V)` is a tag that is used to mark internal structures. It indicates that this list is already being visited.

After drawing the bars, the internal form of the list is processed once more. The atoms `a` and `e` are rendered. The sublist `(b c . d)` is skipped because it is not the last member of the list being visited. A vertical bar is emitted in its place, so it can be rendered later. After displaying an atom, it is replaced with an `(N)` tag, which represents "nothing":

```
((V) a (b c . d) e)   renders    a        |        e
                      gives     ((V) (N) (b c . d)))
```

Note that trailing `(N)`s can be removed immediately. This has been done above.

After drawing another series of bars, the structure is visited once again:

```
((V) (N) (b c . d))  renders  __________[o|o]---[o|o]--- d
                     gives    ((V) (N) ((V) b c))
```

This time the embedded improper list `(b c . d)` is converted to internal representation and its spine is rendered. The trailing `d` of the improper list is removed after displaying it.

The underscores (_) represent the spaces that are used to render "nothing" (`(N)`), so the `(N)` tag makes sure that the embedded spine is properly indented.

After drawing another set of bars, the final iteration finishes the process:

```
((V) (N) ((V) b c))   renders   __________ b        c
                      gives     ((V) (N) ((V)))
```

Because `((V))` is essentially equal to `(N)`:

```
((V) (N) ((V)))
= ((V) (N) (N))
= ((V))
```

`((V))` represents an empty visited list internally, so there is nothing left to render and the program terminates.

The final question to answer is which algorithm to use. Given the above data structure, the algorithm is rather obvious. Here is a rough sketch:

– Draw a set of conses, mark the input list "visited" (`((V) ...)`);
– Draw a set of spaces (for `(N)` members) and bars (for other members);
– Draw a set of objects
  – Draw atoms immediately, replace them with `(N)`;
  – Draw bars for embedded lists except for the last one;
  – If the last object is a list, render its conses and mark it "visited".
– Remove trailing `(N)`s and `((V))`s;
– Repeat steps 2, 3, 4 until the input list is empty (`((V))`).

Of course, this algorithm works only as long as its input it a pair. Atomic input has to be handled separately.

4.2 The DRAW-TREE Program

The *draw-tree* program draws box diagrams of its input, which may be a Scheme atom or pair. It does not support vectors. The program is purely functional and uses side effects only for output. It works by rewriting the internal structure discussed in the previous section.

The first lines of the code already contain a subtle hack that needs some explanation:

```
(define *nothing* (cons 'N '()))

(define *visited* (cons 'V '()))
```

The tags used to represent visited lists and "nothing" in the internal structure impose a problem: how do you render a structure like `((N))`, which contains the "nothing" tag as data? *Draw-tree* renders this input just fine, but how does it do it?

The above definitions of **nothing** and **visited** make use of a subtle side effect which is caused by the following two properties of Scheme:

– `Cons` always returns a *fresh* pair.
– `Eq` returns `#t` only for *identical* objects.

So the formula

```
(eq? (cons x '()) (cons x '())) => #f
```

holds for any value of *x*.

By defining **nothing** and **visited** as fresh pairs and checking them with `eq?`, collisions between **nothing** and `(N)` and **visited** and `(V)` are excluded.

The following predicates check whether an object is ‘‘nothing’’ and whether a list is currently being visited, respectively:

```
(define (empty? x) (eq? x *nothing*))

(define (visited? x) (eq? (car x) *visited*))
```

Mark-visited marks a list as visited, *members-of* returns the members of a list that is being visited:

```
(define (mark-visited x) (cons *visited* x))

(define (members-of x) (cdr x))
```

The *done?* predicate checks whether a (sub)list has been rendered completely:

```
(define (done? x)
  (and (visited? x)
       (null? (cdr x))))
```

The *void* procedure is used to indicate an uninteresting value. It is returned by procedures that are called exclusively for their side effects.

```
(define (void) (if #f #f))
```

The idea behind the *draw-fixed-string* procedure (below) is to print single-character and two-character atoms directly under the bar connecting the atom to a box:

```
[o|o]---[o|o]---[o|/]
 |       |       |
 a       bc     def
```

The procedure draws a string with a fixed length of eight characters. If the string to be printed is longer than seven characters, it is truncated to seven characters. If it is shorter than three characters, a blank is appended in front of it. The resulting string is left-adjusted in a field of eight blanks and emitted.

```
(define (draw-fixed-string s)
  (let* ((b (make-string 8 #\space))
         (k (string-length s))
         (s (if (> k 7) (substring s 0 7) s))
         (s (if (< k 3) (string-append " " s) s))
         (k (string-length s)))
    (display (string-append
              s
              (substring b 0 (- 8 k))))))
```

The *bottom* procedure, which is used by *draw-atom* (below), is not part of Scheme. In fact it is (or should be) *undefined* and serves only as an indicator that something went wrong. If your Scheme environment refuses to run a program containing an undefined symbol, you may include a definition like this one:

```
(define (bottom x) (quotient x 0))
```

Draw-atom converts an atomic Scheme datum into a string and renders it using *draw-fixed-string*.

```
(define (draw-atom n)
  (cond ((null? n)
         (draw-fixed-string "()"))
        ((symbol? n)
         (draw-fixed-string (symbol->string n)))
        ((number? n)
         (draw-fixed-string (number->string n)))
        ((string? n)
         (draw-fixed-string (string-append "\"" n "\"")))
        ((char? n)
         (draw-fixed-string (string-append "#\\"
                                           (string n))))
        ((eq? n #t)
         (draw-fixed-string "#t"))
        ((eq? n #f)
         (draw-fixed-string "#f"))
        (else
         (bottom "draw-atom: unknown type" n))))
```

Draw-conses displays a spine of boxes corresponding to a list. If the list is an improper list, it prints an atom at the end of the spine and otherwise it prints a box containing a slash in the cdr part. For instance:

```
(draw-conses '(a b . c))  displays  [o|o]---[o|o]--- c
(draw-conses '(a b c))    displays  [o|o]---[o|o]---[o|/]
```

The function returns the list that was passed to it.

```
(define (draw-conses n)
  (letrec
    ((d-conses
       (lambda (n)
         (cond ((not (pair? n))
                (draw-atom n))
               ((null? (cdr n))
                (display "[o|/]"))
               (else
                (display "[o|o]---")
                (d-conses (cdr n)))))))
    (d-conses n)
    n))
```

The *draw-bars* procedure traverses a visited list, drawing a bar for each non-`(N)` member and a blank for each `(N)` member. Bars of embedded visited lists are rendered recursively. For instance,

```
(draw-bars `(,*visited* (a b) ,*nothing* (,*visited* c d)))
```

generates the following output (underscores denote spaces):

```
_|_________________|________|_______
```

The first bar will connect to `(a b)`, the gap is caused by `,*nothing*`, the final two bars will connect to `c` and `d` respectively.

```
(define (draw-bars n)
  (letrec
    ((d-bars
       (lambda (n)
         (cond ((not (pair? n))
                (void))
               ((empty? (car n))
                (draw-fixed-string "")
                (d-bars (cdr n)))
               ((and (pair? (car n))
                     (visited? (car n)))
                (d-bars (cdar n))
                (d-bars (cdr n)))
```

```
                (else
                  (draw-fixed-string "|")
                  (d-bars (cdr n)))))))
    (d-bars (members-of n))))
```

Skip-empty is used to skip over empty slots in visited lists, e.g.:

```
(skip-empty `(,*nothing* (,*visited*) (foo))) => ((foo))
```

In combination with *remove-trailing-nothing* it is used to remove trailing empty slots from internal lists:

```
(define (skip-empty n)
  (letrec
    ((skip2
       (lambda (n)
         (cond ((null? n)
                '())
               ((or (empty? (car n))
                    (done? (car n)))
                (skip2 (cdr n)))
               (else
                n)))))
    (skip2 n)))

(define (remove-trailing-nothing n)
  (reverse (skip-empty (reverse n))))
```

The *draw-members* procedure renders the members that are attached to a spine by bars. This includes the spines of embedded lists:

```
Input                            Rendered Output
`(,*visited* a b c))             _a_______b_______c______
`(,*visited* ,*nothing* (a b))   _________[o|o]---[o|/]
`(,*visited* a (b c)))           _a______[o|o]---[o|/]
```

Draw-members also rewrites its input in the following ways: it replaces atoms with `(N)` and marks trailing not-yet-visited lists as visited. It also removes trailing empty slots:

```
Input                              Result
`(,*visited* a b c))               ((V))
`(,*visited* ,*nothing* (a b))     ((V) (N) ((V) a b)))
`(,*visited* a (b c)))             ((V) (N) ((V) b c)))
```

The procedure implements most of the algorithm outlined at the end of the previous section.

```
(define (draw-members n)
  (letrec
    ((d-members
       (lambda (n r)
         (cond ((not (pair? n))
                (reverse r))
               ((empty? (car n))
                (draw-fixed-string "")
                (d-members (cdr n)
                           (cons *nothing* r)))
               ((not (pair? (car n)))
                (draw-atom (car n))
                (d-members (cdr n)
                           (cons *nothing* r)))
               ((null? (cdr n))
                (d-members (cdr n)
                           (cons (draw-final (car n)) r)))
               (else
                (draw-fixed-string "|")
                (d-members (cdr n)
                           (cons (car n) r)))))))
    (mark-visited
      (remove-trailing-nothing
        (d-members (members-of n) '())))))
```

The *draw-final* procedure is called by *draw-members* to render the last slot of a visited list. If it contains an atom, the atom is simply emitted. When the slot contains a list that is already being visited, its members are drawn recursively. When the slot contains a not-yet-visited list, it is tagged visited and its spine is rendered.

Draw-final rewrites the value of the slot in the same way as *draw-members* and returns it.

```
(define (draw-final n)
  (cond ((not (pair? n))
         (draw-atom n)
         *nothing*)
        ((visited? n)
         (draw-members n))
        (else
         (mark-visited (draw-conses n)))))
```

Unsurprisingly, *draw-tree* is the main procedure of the program. It renders a box diagram corresponding to its argument and returns an uninteresting value. The body of its `letrec` handles atomic input and the embedded *d-tree* procedure iterates over input in form of a list until all of its members have been displayed.

```
(define (draw-tree n)
  (letrec
    ((d-tree
       (lambda (n)
         (cond ((done? n)
                (void))
               (else
                (newline)
                (draw-bars n)
                (newline)
                (d-tree (draw-members n)))))))
    (if (not (pair? n))
        (draw-atom n)
        (d-tree (mark-visited (draw-conses n))))
    (newline)))
```

Of course *draw-tree* is a long word to type, so:

```
(define dt draw-tree)
```

The End

In case you arrived here after reading the complete book: Congratulations. You should now be able to read some more serious texts about Scheme. Some titles that may be of interest to you are cited below.

The R^{5}RS is the climax of Scheme standards (so far) and a great improvement over its immediate successor.

R. Kelsey, W. Clinger, J. Rees (eds.)
"Revised^5 Report on the Algorithmic Language Scheme"
Higher-Order and Symbolic Computation,
Vol. 11, No. 1, August, 1998
ACM SIGPLAN Notices, Vol. 33, No. 9, September, 1998
Full text: http://www.schemers.org/Documents/Standards/R5RS/

The following titles occur in order of increasing complexity:

Daniel P. Friedman, Matthias Felleisen
"The Little Schemer"
MIT Press, 1995 (4th Edition)
ISBN 978-0-262-56099-3
Info: http://www.ccs.neu.edu/home/matthias/BTLS/

Hal Abelson, Jerry Sussman, Julie Sussman
"Structure and Interpretation of Computer Programs"
MIT Press, 1996 (2nd Edition)
ISBN 978-0-262-51087-5
Full text: http://mitpress.mit.edu/sicp/

Christian Queinnec
"Lisp In Small Pieces"
Cambridge University Press, 1994
ISBN 978-0-521-54566-2
Info: http://www-spi.lip6.fr/~queinnec/WWW/LiSP.html

Appendix

A.1 First Steps in Scheme
Getting an Interpreter

Because there is a lot of examples to try in this book, the first thing you will want is a Scheme environment. For your first steps you should choose a stable, well-documented, and easy-to-use implementation. Make sure to get an R5RS or R4RS-compliant version. Here are a few suggestions:

PLT Scheme comes in two flavors: **MzScheme** is a command line environment with very good error reporting. For those who cannot live without it: **DrScheme** has an IDE. The 3.x versions of both variants use on the fly compilation. PLT Scheme is open source software with an unrestrictive license. The 2.x and 3.x versions are fine.
URL: `http://www.plt-scheme.org/`

Scheme 48 is a very clean implementation that is quite picky about the programs that it accepts (this is a good thing). It is free software.
URL: `http://www.s48.org/`

Scheme 9 from Empty Space is a free, small, and portable implementation that compiles out of the box on a wide variety of systems. It implements only a subset of R4RS, though.
URL: `http://www.t3x.org/bits/s9fes/`

If you prefer the big picture, here is a collection of lots of implementations. Have a look and see for yourself:
`http://community.schemewiki.org/?scheme-faq-standards#implementations`

The FreeBSD Project's ports collection has a Scheme section, too:
`http://www.freebsd.org/ports/scheme.html`

Running the Interpreter

Once you have chosen an environment, installed it, and launched it, it will greet you with some kind of banner:

```
Foo Scheme ready
> _
```

At this point you can type in or paste an expression. When the expression has a value, the environment will print it and prompt you for another expression:

```
Foo Scheme ready
> (cons 1 2)
(1 . 2)
> _
```

When something goes wrong, an informative message will print:

```
> (car 'x)
car: expected a pair, but got: x
> _
```

An interactive environment is also called a *read eval print loop* (or in short: REPL), because this is what it does: read an expression, evaluate it, print the result, and loop.

To end a session it is normally sufficient to type an end-of-file character such as `control-D` or `control-Z`. If this does not work, typing `(exit)` or `(quit)` is worth a try.

Testing Programs

You will not want to enter programs with a size of more than two or three lines directly at the REPL. Something you definitely want is a decent programmer's editor. `Vi` or `Emacs` are fine, if you are used to them. In fact, *any* editor will do, but one thing that *really* helps is the capability to show matching parentheses, so you should get an editor that can do this. When using `vi`, you can use the `:set showmatch` option and/or the `%` command.

To test a program, launch your editor in one command line window and the Scheme environment in another. Make sure that your editor saves to the directory in which you started Scheme.

Key in your program in the editor window and save the text when you are done. Here is a sample program to try:

```
(define (hello)
  (display "Hello, World!")
  (newline))
```

After keying in the program text, save the file. Scheme program files normally have a `.scm` suffix, so you might name your source file `hello.scm`, for example.

To load the program into the Scheme environment, type

```
(load "hello.scm")
```

at the REPL. You can now run the program by calling the *hello* function:

```
> (hello)
Hello, World!
> _
```

Above example program does not have a specific result. Some environments print unspecific results, some omit them.

If you change the program in your editor window, you have to re-load it at the REPL in order to transfer the changes to the Scheme environment.

Should your program "hang" during execution, you can normally hit `control-C` (or whatever key interrupts program execution on your system) to stop it and return to the REPL.

Batch Mode

Maybe you are used to working on a command line and prefer to run programs in batch mode. Most Scheme implementations can do this, too. See the documentation for the actual command line options that are required to do so. Because there is no interaction in batch mode, you have to include the function application that starts your program in the program file, so you would have to change above example to:

```
(define (hello)
  (display "Hello, World!")
  (newline))

(hello) ; start the program
```

You can then run the program from the shell prompt or DOS prompt using a command like this:

```
your-scheme -f hello.scm
```

Of course, you will have to replace `your-scheme` with the name of your Scheme system and `-f` with the option that actually loads and runs a program in batch mode.

In batch mode no greeting banner is printed and the environment does not enter the REPL. All expressions of the specified program are evaluated and then the Scheme process terminates. In case of an error or keyboard interrupt, the Scheme process prints an error message and terminates immediately.

When your Scheme system offers a stand-alone compiler, you can compile Scheme programs to executables and run them just like programs written in other languages. Many systems offer both an interactive environment and a stand-alone compiler.

Note that a REPL is not a reliable indicator for interpretive program execution. There are more and more systems that compile expressions entered at the REPL on the fly and execute them as native code, thereby combining the convenience of an interactive environment with the speed of compiled code.

A.2 Scheme Style Guide

The rules and templates listed in this appendix are useful for writing readable Scheme programs. They are intended to reflect the logical structure of programs. Of course, *style* is a highly personal matter, so feel encouraged to develop your own one.

Conventions are there to be bent or broken occasionally. When the rules get in your way, feel free to ignore them. Make your *code* as readable as you can. Your code is a document written in an abstract language that describes how to solve a specific problem. It just happens to be ''executable''.

Here are the most basic rules for formatting Scheme programs:

- Use a monospace font for editing Scheme code;
- Use blanks rather than TABs for indentation;
- Indentation of code is done in steps of two characters;
- Indentation of data is done in steps of one character;

– One level of indentation is added at each opening parenthesis;
– One level of indentation is subtracted at each closing parenthesis;
– Subsequent closing parentheses stick together.

The following program would be acceptable, but not really beautiful:

```
(define (foo x)
  (and
    (pair?
      x)
    (pair?
      (cdr
        x))
    (cadr x)))
```

This is why there are two additional rules:

– Whenever a subexpression fits in a single line without obscuring meaning too much, it should be placed in a single line.

```
(define (foo x)
  (and
    (pair? x)
    (pair? (cdr x))
    (cadr x)))
```

– When sufficient horizontal space exists, the first argument of a function or pseudo function application should be placed in the same line as the function itself and the remaining arguments should start in the same column as the first one.

```
(define (foo x)
  (and (pair? x)
       (pair? (cdr x))
       (cadr x)))
```

These simple rules are sufficient for writing quite readable code. Further details and exceptions follow below.

A.2.1 Definitions and Bindings

Most variable definitions are one-liners:

```
(define variable value)
```

Variable definitions with lengthy values (like procedures) have their values indented:

```
(define variable
  (lambda (formal ...)
    body))
```

The following style is preferred for static procedures (procedures whose names are not assigned different values at a later time):

```
(define (procedure formal ...)
  body)
```

Very short procedures can be defined in single lines:

```
(define (procedure formal) body)
```

The structure of `define-syntax` and `syntax-rules` is rather static:

```
(define-syntax keyword
  (syntax-rules (symbol ...)
    (pattern template)
    ...
    (pattern template)))
```

or:

```
(define-syntax keyword
  (syntax-rules (symbol ...)
    (pattern
      template)
    ...
    (pattern
      template)))
```

Both patterns and templates of syntax rules are formatted like code, not like data, e.g.:

```
(define-syntax when
  (syntax-rules ()
    ((_ predicate consequent ...)
       (if predicate
           (begin consequent
                  ...)))))
```

Short lambda functions are placed in a single line:

```
(lambda (formal ...) body)
```

Otherwise the body is indented:

```
(lambda (formal ...)
  body)
```

`Let` and `let*` both follow the same scheme. Their first arguments are indented like data at the first level of nesting, so the opening parentheses in front of the variables are in the same column:

```
(let ((variable value)
      ...
      (variable value))
  body)
(let* ((variable value)
       ...
       (variable value))
  body)
```

When a value of `let` or `let*` does not fit in a single line, it is indented as usual:

```
(let ((s (cond ((positive? x) "positive")
               ((negavive? x) "negative")
               (else          "zero"))))
  (string-append "x is " s))
```

When horizontal space is tight, `letrec`-style indentation may be used.

Because `letrec` is mostly used to define procedures, `letrec` itself and the symbols bound by it are placed in separate lines to save horizontal space:

```
(letrec
  ((symbol
     (lambda (formal ...)
       body))
   ...
   (symbol
     (lambda (formal ...)
       body)))
  letrec-body)
```

A.2.2 Procedure Application

There are not many choices for procedures of no arguments:

```
(procedure)
```

Applications that fit in a single line are placed in a single line:

```
(procedure argument ...)
```

In longer expressions, the arguments are lined up with respect to a common column:

```
(procedure argument
           ...
           argument)
```

If there is insufficient horizontal space, arguments may be indented in the lines following the procedure:

```
(procedure
  argument
  ...
  argument)
```

Short lambda function applications are placed in a single line:

```
((lambda (variable ...) body) argument ...)
```

The arguments of more complex applications are indented by *one* blank so that they share the first column with the beginning of the lambda function itself:

```
((lambda (variable ...)
   body)
 argument
 ...
 argument)
```

You will rarely need this, though, unless you are dealing with lambda calculus.

A.2.3 Conditionals, Logic, Etc

Applications of `and`, `begin`, `if`, and `or` are preferably indented in the

following way (but short applications may be compacted to single lines):

```
(and expression
     ...
     expression)

(begin expression
       ...
       expression)

(if predicate          (if predicate
    consequent             consequent)
    alternative)

(or expression
    ...
    expression)
```

The difference between the following `case` styles is mostly a matter of taste (the left one may save some horizontal space, though):

```
(case expression    (case expression
  (list body)             (list body)
  ...                     ...
  (list body))            (list body))
```

When the lists of cases are long or have different lengths, the following style may be preferable:

```
(case expression
  (list
    body)
  ...
  (list
    body))
```

When all lists of cases have similar lengths, the bodies may be adjusted with respect to a common column:

```
(case digit
      ((1 3 5 7 9) "odd")
      ((0 2 4 6 8) "even")
      (else        "not a digit"))
```

The formatting of `cond` is similar to that of `case`. This style is used for

short precidates and bodies:

```
(cond (predicate body)
      ...
      (predicate body))
```

In `cond` expressions with long precicates and/or bodies the following format saves some horizontal space:

```
(cond (predicate
        body)
      ...
      (predicate
        body))
```

This format squeezes out four more characters:

```
(cond
  (predicate
    body)
  ...
  (predicate
    body))
```

Even in `cond` two left-adjusted columns may be used for predicates and bodies:

```
(cond ((< x 0) "negative")
      ((> x 0) "positive")
      (else    "zero"))
```

A.2.4 Data and Quotation

When quoted lists fit in a single line, this is how it should be done:

```
'(member ...)
(quote(member ...))
```

Otherwise members are indented by a *single character*:

```
'(member           (quote (member
  ...                       ...
  member)                   member))

'((member ...)     (quote ((member ...)
  ...                       ...
  (member ...))             (member ...)))
```

The following template illustrates why indentation by a single character is essential, especially in deeply nested lists:

```
'(((member ...)
   ...
   (member ...))
  ((member ...)
   ...
   (member ...)))
```

Although quasiquoted forms are technically data, they are mostly used to form expressions, so they are formatted in the style of code:

```
`(if ,predicate
     ,consequent
     ,alternative)

`(lambda ,formals
  ,@body)
```

A.3 Table of Example Programs

`add`	add lists of digits	82
`append2`	append lists, tail recursive version	30
`case`	`case` syntax	115
`checked-len`	`length` with non-local exit	127
`complement`	negate predicates	37
`compose`	function composition	57
`copy`	copy files	106
`depth`	depth of lists	50
`depth`	depth of lists, using HOFs	72
`draw-tree`	draw box diagrams of Scheme data	148
`eql?`	equality of lists of symbols	62
`equal?`	general equality predicate	64
`expand-qq`	quasiquote expander	119
`explode`	explode symbols	80
`fact`	factorial function	11
`fact`	factorial function, tail recursive version	18
`filter`	extract list elements	37
`implode`	implode lists of symbols	81
`int->list`	convert integer to list	82

`kons`	`cons` as a combinator	73
`list->int`	convert list to integer	83
`map`	map procedures over lists, variadic version	70
`mapcar`	map procedures over lists	68
`max`	maximum of arguments	54
`max-list`	maximum of a list	50
`modulo`	remainder of integer division	87
`permute`	permutations of strings	96
`quine`	a Scheme quine	118
`read-line`	read lines of characters	105
`remove`	remove list elements	37
`reverse`	reverse lists	76
`reverse-syn`	reverse lists using syntax	114
`rotations`	rotations of a string	93
`sub-strings`	substrings of a string	91
`type`	copy files to current output port	106
`when`	`when` syntax	109
`Y`	Y combinator	139

A.4 Scheme Syntax and Procedures

This is a summary of the Scheme syntax and procedures discussed in this book.

Key

`body`	a sequence of expressions	`proc`^{n}	a procedure of *n* arguments
`char`	a char	`str`	a string
`expr`	any type	`sym`	a symbol
`form`	any datum	`tval`	a truth value
`int`	an integer	`void`	an unspecific value
`list`	a list	`[x]`	*x* is optional
`num`	any numeric type	`x ...`	zero or more occurrences of *x*
`pair`	a pair	`x|y`	*x* or *y*
`proc`	a procedure		

Syntax

`'form => form`

Turn *form* into a datum. Alias: `quote`.

`,expr => form`

Evaluate *expr* in a `quasiquote` template. Alias: `unquote`.

`,@expr => spliced-form`

Evaluate *expr* in a `quasiquote` template and splice into surrounding list. Alias: `unquote-splicing`.

`` `qq-template => form ``

Quasiquote the quasiquote template *qq-template*. Alias: `quasiquote`.

`(`$\texttt{expr}_0$ $\texttt{expr}_1$ `...)`

Apply $expr_0$ to $expr_1$... $Expr_0$ must reduce to a procedure.

`(and` $\texttt{expr}_1$ `...) => form`

Evaluate expressions in sequence until one gives `#f`.

`(begin` $\texttt{expr}_1$ `...) => form`

Evaluate expressions in sequence.

`(case expr (`$\texttt{list}_1$ $\texttt{body}_1$`) (`$\texttt{list}_2$ $\texttt{body}_2$`) ...) => form`

Evaluate first *body* whose *list* contains *expr*.

`(cond (`$\texttt{expr}_1$ $\texttt{body}_1$`) (`$\texttt{expr}_2$ $\texttt{body}_2$`) ...) => form`

Evaluate first *body* whose *expr* reduces to truth.

`(define sym expr) => void`

Introduce *sym* and bind it to (a location referring to) *expr*.

`(define (`$\texttt{sym}_1$ $\texttt{sym}_2$ `...) body) => void`

`(define (`$\texttt{sym}_1$ $\texttt{sym}_2$ `... .` $\texttt{sym}_n$`) body) => void`

Introduce sym_1 and bind it to (a location referring to) a procedure with the variables sym_2... and the given *body*.

`(define-syntax keyword transformer) => void`

Introduce the keyword *keyword* and bind it to the syntax transformer *transformer*. See also: `syntax-rules`.

`(if` $\texttt{expr}_1$ $\texttt{expr}_2$ `[`$\texttt{expr}_3$`]) => form`

Evaluate to $expr_2$ if $expr_1$ reduces to truth, and to $expr_3$ otherwise.

`(lambda sym body) => proc`

`(lambda (`$\texttt{sym}_1$ $\texttt{sym}_2$ `... .` $\texttt{sym}_n$`) body) => proc`

Evaluate to a variadic procedure with the variable *sym* (or the variables sym_1...) and the given *body*.

`(lambda (sym1 ...) body) => proc`

Evaluate to a procedure with the variables *sym1*... and the given *body*.

`(let ((sym1 expr1) ...) body) => form`

Bind *symi* to *expri* locally giving a new context and evaluate *body* in that context.

`(let* ((sym1 expr1) ...) body) => form`

Like `let`, but bind expressions to symbols sequentially.

`(letrec ((sym1 expr1) ...) body) => form`

Like `let`, but fix recursive references.

`(or expr1 ...) => form`

Evaluate expressions in sequence until one gives truth.

`(quasiquote qq-template) => form`

Quasiquote the quasiquote template *qq-template*. Alias: `` ` ``.

`(quote form) => form`

Turn *form* into a datum. Alias: `'`.

`(set! sym expr) => void`

Change the value of *sym* to *expr*.

`(syntax-rules (sym1 ...) (pat1 temp1) (pat2 temp2) ...) => transformer`

Evaluate to a syntax transformer. *Sym1*... are local keywords, *pat* is a pattern and *temp* is a template. See also: `define-syntax`.

`(unquote expr) => form`

Evaluate *expr* in a `quasiquote` template. Alias: `,`.

`(unquote-splicing expr) => spliced-form`

Evaluate *expr* in a `quasiquote` template and splice into surrounding list. Alias: `,@`.

Procedures

`(* num1 ...) => num`

Evaluate to the product of the given numbers.

`(+ num1 ...) => num`

Evaluate to the sum of the given numbers.

`(- num1 num2 ...) => num`

Evaluate to the difference of the given numbers. If only one argument *n* is given, return `-n`.

`(/ num1 num2 ...) => num`

Evaluate to the quotient of the given numbers. If only one argument *n* is given, return `1/n`.

`(< num1 num2 num3 ...) => tval`
`(<= num1 num2 num3 ...) => tval`
`(> num1 num2 num3 ...) => tval`
`(>= num1 num2 num3 ...) => tval`

Check whether the given arguments are in a specific order. E.g., `<` checks whether its arguments are in strict ascending order.

`(= num1 num2 num3 ...) => tval`

Check whether the given numbers are equivalent.

`(abs num) => num`

Return the absolute value of *num*.

`(append list1 ...) => list`

Append lists, giving a (proper) list.

`(append list1 ... expr) => pair`

Append lists, giving a (potentially improper) list.

`(apply proc expr1 ... list) => form`

Apply *proc* to the arguments in list. If *expr*s are given, cons them to *list*.

`(assoc expr alist) => pair|#f`

Find pair with key=*expr* in *alist*. Use `equal?` to compare keys.

`(assq expr alist) => pair|#f`

Find pair with key=*expr* in *alist*. Use `eq?` to compare keys.

`(assv expr alist) => pair|#f`

Find pair with key=*expr* in *alist*. Use `eqv?` to compare keys.

`(caar pair) ... (cddddr pair) => form`

Extract members of lists. E.g. `(cadr x)` = `(car (cdr x))`.

`(call-with-current-continuation proc1) => form`
`(call/cc proc1) => form`

Capture current continuation and pass it to *proc*[1].

`(car pair) => form`

Extract car part (head) of pair.

`(cdr pair) => form`

Extract cdr part (tail) of pair.

`(char-ci<? char1 char2 char3 ...) => tval`
`(char-ci<=? char1 char2 char3 ...) => tval`

continued...

`(char-ci>? char1 char2 char3 ...) => tval`
`(char-ci>=? char1 char2 char3 ...) => tval`

Check whether the given characters are in a specific lexical order. E.g., `char-ci<?` checks whether its arguments are in strict alphabetical order. Ignore the case of the characters.

`(char-ci=? char1 char2 char3 ...) => tval`

Check whether the given characters are equal. Ignore their case.

`(char->integer char) => int`

Convert char to integer.

`(char-alphabetic? char) => tval`
`(char-lower-case? char) => tval`
`(char-numeric? char) => tval`
`(char-upper-case? char) => tval`
`(char-whitespace? char) => tval`

Check properties of characters. E.g. `(char-numeric? x)` checks whether *x* is in the range `#\0...#\9`.

`(char-downcase char) => char`
`(char-upcase char) => char`

Change the case of a letter. E.g. `(char-upcase #\c) => #\C`.

`(char<? char1 char2 char3 ...) => tval`
`(char<=? char1 char2 char3 ...) => tval`
`(char>? char1 char2 char3 ...) => tval`
`(char>=? char1 char2 char3 ...) => tval`

Check whether the given characters are in a specific lexical order. E.g., `char<?` checks whether its arguments are in strict alphabetical order.

`(char=? char1 char2 char3 ...) => tval`

Check whether the given characters are equal.

`(cons expr1 expr2) => pair`

Construct a fresh pair with head=*expr*$_1$ and tail=*expr*$_2$.

`(display expr [out-port]) => void`

Print *expr* in a prettier but less accurate form than `write`. If an output port is specified, write to that port.

`(eof-object? expr) => tval`

Check whether *expr* is the end-of-file object.

`(eq? expr1 expr2) => tval`

Check whether *expr*$_1$ and *expr*$_2$ are identical.

`(equal? expr1 expr2) => tval`

Check whether *expr*$_1$ and *expr*$_2$ are equal.

`(eqv? expr`$_1$ `expr`$_2$`) => tval`
Check whether *expr*$_1$ and *expr*$_2$ are equivalent.

`(floor num) => int`
Return the greatest integer that is not greater than *num*.

`(gcd int`$_1$ `...) => int`
Return the greatest common divisor of *int*$_1$....

`(integer->char int) => char`
Convert integer to char.

`(lcm int`$_1$ `...) => int`
Return the least common multiple of *int*$_1$....

`(length list) => int`
Return the length of *list*.

`(list expr`$_1$ `...) => list`
Create a fresh list containing the given *expr*s and return it.

`(list->string list) => str`
Convert list to string.

`(list-ref list int) => form`
Extract the *int*'th member of *list*.

`(list? expr) => tval`
Check whether *expr* is a (proper) list.

`(map proc list`$_1$ `list`$_2$ `...) => list`
Map procedure *proc* over the given *list*s.

`(max num`$_1$ `num`$_2$ `...) => num`
Return the greatest of the given numbers.

`(member expr list) => list|#f`
Return the first sublist of *list* whose head equals *expr*, or `#f` if no such sublist exists. Use `equal?` to compare list elements.

`(memq expr list) => list|#f`
Return the first sublist of *list* whose head equals *expr*, or `#f` if no such sublist exists. Use `eq?` to compare list elements.

`(memv expr list) => list|#f`
Return the first sublist of *list* whose head equals *expr*, or `#f` if no such sublist exists. Use `eqv?` to compare list elements.

`(min num`$_1$ `num`$_2$ `...) => num`
Return the least of the given numbers.

`(modulo int`$_1$ `int`$_2$`) => int`
Return *int*$_1$ modulo *int*$_2$ (division remainder).

`(negative? num) => tval`
Check whether *num* is negative.

`(newline [out-port]) => void`

Display a newline sequence. If an output port is given, write to that port.

`(not expr) => tval`

Logical not, i.e. `(eq? expr #f)`.

`(null? expr) => tval`

Check whether *expr* is the empty list.

`(number? expr) => tval`

Check whether *expr* is a number.

`(pair? expr) => tval`

Check whether *expr* is a pair.

`(quotient int1 int2) => int`

Return the integer part of the quotient of int_1 and int_2.

`(read [in-port]) => form`

Read the external representation of a form and return a corresponding internal representation. If in input port is given, read from that port.

`(read-char [in-port]) => char`

Read a character and return it. If in input port is given, read from that port.

`(remainder int1 int2) => int`

Return the division remainder of int_1 and int_2.

`(sqrt num) => num`

Evaluate to the square root of *num*.

`(string char1 ...) => str`

Create a fresh string containing the given *char*s and return it.

`(string->list str) => list`

Convert string to list (of chars).

`(string->symbol str) => sym`

Convert string to symbol.

`(string-ci<=? str1 str2 str3 ...) => tval`
`(string-ci<? str1 str2 str3 ...) => tval`
`(string-ci>=? str1 str2 str3 ...) => tval`
`(string-ci>? str1 str2 str3 ...) => tval`

Check whether the given strings are in a specific lexical order. E.g., `string-ci<?` checks whether its arguments are in strict alphabetical order. Ignore case.

`(string-ci=? str1 str2 str3 ...) => tval`

Check whether the given strings are equal. Ignore case.

`(string-length str) => int`

Return the length of a string.

`(string-ref str int) => char`

Return *int*'th char of *str*.

`(string<=? str`$_1$ `str`$_2$ `str`$_3$ `...) => tval`

`(string<? str`$_1$ `str`$_2$ `str`$_3$ `...) => tval`

`(string>=? str`$_1$ `str`$_2$ `str`$_3$ `...) => tval`

`(string>? str`$_1$ `str`$_2$ `str`$_3$ `...) => tval`

Check whether the given strings are in a specific lexical order. E.g., `string<?` checks whether its arguments are in strict alphabetical order.

`(string=? str`$_1$ `str`$_2$ `str`$_3$ `...) => tval`

Check whether the given strings are equal.

`(substring str int`$_1$ `int`$_2$`) => str`

Extract substring starting at int_1 and extending up to but not including int_2 from *str* and return it.

`(symbol->string sym) => str`

Convert symbol to string.

`(with-input-from-file str proc`^{0}`) => form`

Connect the standard input port to the file named in *str* and evaluate $(proc^0)$ in that context.

`(with-output-to-file str proc`^{0}`) => form`

Connect the standard output port to the file named in *str* and evaluate $(proc^0)$ in that context.

`(write expr [out-port]) => void`

Write the external representation of *expr*. If an output port is specified, write to that port.

`(zero? num) => tval`

Check whether *num* equals zero.

Index

#

#f 22
#t 22
#\newline 24
#\space 24
' 27, 166
() 45
* 84, 14, 66
+ 84, 10, 66
, 116, 167
,@ 117, 167
-> 13
- 84, 10
... 111
/ 84
< 59
<= 59
= 59
=> 9
> 50
>= 59

` 116, 167

A

alist 46
alpha conversion 141
and 63, 121, 164
append 29
applicative language 100
applicative order 143
apply 56, 69
argument 9
assoc 47
association list 46
assq 67
assv 67
atom 44

B

begin 103, 121, 164
beta reduction 139
bignum arithmetics 22, 82
binding 12, 26, 111
boolean 22
bottom 10, 150
bottom 54
bound variable 34
box notation 145
boxed value 75

C

caaaar...cddddr 48
cadr 36
call by name 56, 143
call by value 56, 143
call/cc 125
call-with-current-continuation 125
car 28, 51
case 114, 165
case insensitive 61
cdr 28, 51
character predicates 60
char>? 77
char<=? 77
char-alphabetic? 88
char-downcase 89
char-lower-case? 88
char-numeric? 88
char-upcase 89

`char-upper-case?` 88
`char-whitespace?` 88
character 23
clause 12, 115
close over 36
closure 32
code 160
combinator 133
comment 9
`cond` 12, 120, 106, 165
`cons` 28, 45
cons object 45, 145
continuation 125
current continuation 125
currying 135
D
default input port 105
`define` 11, 161
`define-syntax` 109, 162
disjointness of types 78
`display` 102, 24
dotted pair 45
dynamic scoping 36
dynamic context 129
dynamic typing 75
E
effect (of a function) 100
ellipsis 111
`else` (of `cond`) 19
empty list 29, 45
eof object 106
`eof-object?` 106
`eq?` 27, 11, 58
`equal?` 64
equality 62, 57
`eqv?` 62
eta conversion 138
evaluation 10
expression 9
external representation 23, 101
F
file 105
first class object 128
fixed point 134
`floor` 86
free variable 34
freelist 97
full numeric tower 88
function 9
function body 12, 104
G
garbage collection 98
`gcd` 87
H
head (of a list) 29
head (of a procedure) 12
higher order function 68
I
identity 57, 27
`identity` 128
`if` 20, 72, 164
imperative programming 11
improper list 46
indefinite extent 128
indirect recursion 123
inner context 33
integer 21
K
keyword 115
L
`lambda` 31, 54, 73, 163
lambda calculus 140, 164
`lcm` 87
`length` 49
`let*` 41, 122, 163
`let` 33, 91, 122, 163

`letrec` 41, 91, 18, 122, 131, 163
lexical environment 36
lexical scoping 36
LISP 28
list 24
`list->string` 80
`list` 55
`list-ref` 49
`list?` 78

M

`map` 69
`max` 53
`member` 22
memory compation 99
memory fragmentation 99
`memq` 67
`memv` 67
metaprogramming 118
`modulo` 85
mutual recursion 42, 123

N

`negative?` 87
`newline` 104
non-local exit 127
normal form 14
normal order 143
`not` 67
`null?` 29
`number?` 37
numeric predicate 59
numeric tower 88

O

`or` 65, 121, 164
order of evaluation 40, 130
outer context 33

P

pair 44
`pair?` 78
pattern 110
permutation 91
port 105
predicate 12
predicate (of `cond`) 12
primitive function 32
procedure 9
procedure application 9, 164
pseudo functions 109, 11

Q

`quasiquote` 116, 167
quine 118
quotation 25, 166, 116
`quote` 25, 166
`quotient` 85, 10

R

rational value 85
`read-char` 104
read eval print loop 158
`read` 101, 23
recursion 15, 114
reduce 10
`remainder` 85

S

self application 74, 133
`set!` 43
side effect 100, 148, 43
signature 55
spine 146
`sqrt` 88
strict function 10
`string->list` 80
`string->symbol` 81
`string-ci<?` 61
`string-ci=?` 61
`string` 81
`string-length` 91
string predicate 61

`string-ref` 92, 82
`string=?` 61
string 24
style 160
`substring` 89
symbol 24
`symbol->string` 80
syntax 109, 11
`syntax-rules` 109, 162
syntax transformation 109

T

tail (of a list) 29
tail call 120
tail call optimization 17, 123
tail recursive 17, 120
truth value 12
type checking 64
type conversion 80
type predicate 64

U

`unquote` 116, 167
`unquote-splicing` 117, 167
unspecific value 101

V

value 14
variable 12
variadic procedure 53

W

`with-input-from-file` 105
`with-output-to-file` 106
`write` 101, 23

Y

Y combinator 134

Z

`zero?` 12