If you are already familiar with Scheme or another dialect of LISP, you may wish to skim this chapter quickly, taking note of any unfamiliar terminology (indicated by *italics*), and later refer to this chapter for specific information on Scheme. We introduce only those features of Scheme that are used later in this book.

1.1 Simple Expressions

EoPL-1

A *statement* is a programming language construct that is evaluated only for its effect. Examples include assignment statements, input/output statements, and control statements (`while` loops, `if` statements, etc.). Programs in most languages are composed primarily of statements; such languages are said to be *statement oriented.*

Programming language constructs that are evaluated to obtain values are called *expressions.* Arithmetic expressions are the most common example. Expressions may occur as parts of statements, as in the right-hand side of an assignment statement. The data that may be returned as the values of expressions constitute the *expressed values* of a programming language. Expressions that are evaluated solely for their value, and not for any other effects of the computation, are said to be *functional.*

Some programming languages, such as Scheme, are *expression oriented:* their programs are constructed of definitions and expressions; there are no statements. This section reviews basic techniques for constructing expressions in Scheme.

1.1.1 Literals, Procedure Calls, and Variables

The simplest form of expression is a *literal* (or *constant*), which always returns the indicated value. For example, the result of evaluating the *numeral* 2 is a value denoting the number two, which has the printed representation 2. Other literals we shall have occasion to use include *strings,* such as `"This is a string."`, the *boolean* values `#t` (true) and `#f` (false), and *characters,* such as `#\a` and `#\space`. We discuss these and other Scheme data types in the next section.

The next simplest form of expression is a *variable reference.* The value of a variable reference is the value currently associated with, or *bound to,* the variable. A variable is said to *denote* the value of its *binding.* The data that can be bound to variables constitute the *denoted values* of a programming language. Since all variable references in Scheme are also expressions, and

the value of any expression may be bound to a variable, the denoted values and the expressed values of Scheme are the same, at least in the absence of variable assignment (section 4.5).

Variables are represented by *identifiers.* As in most languages, sequences of letters and digits (not starting with a digit) may be used as identifiers, for example: `x`, `x3`, `foo`, and `longidentifier`. Scheme is more permissive than most languages in the use of special characters to form identifiers. For example, the following are all identifiers: `+`, `/`, `two+three`, `zero?`, `long_identifier`, `an-even-longer-identifier`. Some special characters, such as parentheses and spaces, are not allowed in identifiers. Digits may generally be used in identifiers, *e.g.* `x3`, but not as the first character. A few identifiers, such as `define` and `if`, are reserved for use as *keywords* and should generally not be used as variables.

Scheme provides standard bindings for a number of variables. For example, `+` is bound to the addition procedure and `zero?` is bound to a boolean procedure, or *predicate,* that tests whether its argument is zero. Other standard bindings will be introduced as they are needed. We call procedures that are the values of standard bindings *standard procedures.* (See appendix I.)

If a value is the binding of some variable, it is often convenient to refer to the value by the name of the variable. However, the distinction between the name of a variable and the value of its binding is very important. In this book we observe this distinction by using different fonts. When referring to the variable named "x" as a part of a program, we use the standard typewriter-style font: `x`. When referring to the value of the variable `x`, we use an italic font: *x*. Thus we use "*zero?*" instead of "the value of the variable `zero?`" when referring to the numeric zero predicate.

Statement-oriented languages usually distinguish between *functions,* which return values and are used in expressions, and *procedures,* which do not return values and are invoked by *procedure call* statements. Though function calls and procedure calls often look the same, syntactically they are distinct: function calls are expressions, while procedure calls are statements. However, since Scheme does not have statements, it does not make this distinction. In fact, Scheme functions are usually called procedures, and Scheme function calls are then referred to as procedure calls. We use the term "function" to refer only to abstract mathematical functions.

The syntax of procedure calls in Scheme is not typical of other programming languages. For example, a call to the procedure *p* with arguments 2 and 3 is written in Scheme as `(p 2 3)`, instead of `p(2,3)`. Parentheses surround the entire procedure call, and its components are separated by spaces. We say that the procedure *p* is *applied* to the arguments 2 and 3. Procedure (or function) calls are sometimes referred to as *applications* or *combinations.*

The general syntax of procedure calls is

(*operator operand*$_1$ ... *operand*$_n$)

The ellipsis "..." indicates possible repetition. There may in general be any number of operands, or possibly none at all ($n = 0$). The *operator* and each *operand* are components that are themselves expressions. They are called *subexpressions.* The operator subexpression is evaluated to obtain a procedure, while the operand subexpressions are evaluated to obtain the *arguments* of the call before invoking the procedure. (Arguments are also referred to as *actual parameters,* or simply *parameters.*) In Scheme, the order in which the operator and operand subexpressions are evaluated is not specified, but in some languages it is guaranteed to be left to right and in others it is always right to left.

Any expression may be used as an operand in a procedure call. For example, the procedure call

```
(+ x (p 2 3))
```

contains the operand `(p 2 3)`, which is itself a procedure call. If the value of `(p 2 3)` is 6 and *x* is 3, then `(+ x (p 2 3))` is 9. (More precisely, "the value of `(+ x (p 2 3))` is 9." Since a compound expression's value is not likely to be confused with the expression itself, in such cases we shall often omit the phrase "the value of.")

Operators may also be arbitrarily complex, as long as they return procedures. Thus if *g* were a procedure that when applied to 2 returned the addition procedure, then

```
((g 2) 3 4)
```

would return 7. Procedures that return procedures are called *higher-order procedures,* and expressions that return procedures are called *higher-order expressions.* They may be unfamiliar, but much will be accomplished with them later.

1.1.2 Definitions, Programs, and the Read-Eval-Print Loop

Most operations can be expressed as procedure calls. For those that cannot, a small number of *special forms* are required.

Consider the operation of binding the variable `x` to 3. We would like to accomplish this by saying

```
(define x 3)
```

The general definition syntax we use is

`(define` *variable expression*`)`

where *variable* and *expression* indicate an arbitrary variable and expression. If this were a procedure call, with the variable `define` bound to some procedure, *variable* would be evaluated as an argument and its value passed to the procedure. But *variable* may be unbound, in which case it cannot be evaluated. Even if the variable were already bound, say `x` was bound to 7, it still would not do to evaluate the variable. The special form `define` must modify the binding of `x`, which would not be possible if it were simply passed the value 7.

The solution is to declare that the above syntactic form is special—distinct from a procedure call. Each special form is indicated by an identifier, in this case `define`, that should not be used as a variable. These special form identifiers are called *keywords.* Each special form has its own sequencing rule, that is order of evaluation of subexpressions. In this case *expression* is evaluated first, and then *variable* is bound to the value of *expression.*

A Scheme program consists of a sequence of definitions and expressions that are executed in order by the Scheme system. These definitions and expressions are said to be at *top level.* We next discuss a few features of typical *programming environments* in which the Scheme language is used. It should be borne in mind that these are not features of the language itself.

Programs may be stored in a file for convenient loading, or they may be entered interactively. In the *interactive* mode, you enter a definition or expression, which is evaluated as soon as it is complete. When an expression is entered, its value is printed. The system then prints an input prompt and the cycle repeats. This repetitive action is often called the *read-eval-print loop.* The transcript of a brief interactive session follows.

```
> 3
3
> *    ; evaluates to the standard multiplication procedure
#<Procedure>
> (* 2 3)
6
> (define x 3)
> x
```

```
3
> (+ x (* 2 3))
9
```

In this case the Scheme prompt is ">." A semicolon ";" and anything following it on the same line is ignored by Scheme so that comments may be inserted in programs and transcripts. In general, procedures cannot be printed. Thus the system simply prints some indication that a procedure has been returned. In this book "`#<Procedure>`" is that indicator.

Following a definition, many Scheme systems print the name of the variable defined. As the transcript illustrates, however, we choose not to print anything following a definition. This emphasizes that, in general, definitions do not have values. In this respect they are like statements, but their use is more limited. In this book `define` is used only at top level.

A final note about definitions: the value of a variable may be *redefined*. That is, the value of an already defined variable may be changed with another definition.

```
> (define x 2)
> x
2
> (define x (+ 1 x))
> x
3
```

Redefinition is allowed simply to make software development more convenient. In Scheme the values of variables with standard bindings, such as +, can be redefined. This is occasionally useful, for example, if you wish to keep track of how many times + is invoked with a negative argument. Redefinition of standard procedures, however, is risky; others may depend on them in unexpected ways.

The interactive nature of Scheme aids program development. It is also helpful in learning Scheme, because it makes it easy to try things out if you wish to test your understanding or discover what will happen. Transcripts of interactions with Scheme are also a convenient way of providing examples. We use them frequently. You are urged to study our examples carefully to be sure you understand why Scheme behaves as it does. Sometimes definitions made in one transcript will be used in other transcripts that follow.

• *Exercise 1.1.1*
Start interacting with Scheme today! □

Read-eval-print loops and redefinitions may not be appropriate in some programming environments. For example, a Scheme implementation might be designed to compile Scheme programs on one machine for execution at a later time on other machines. In this case a read-eval-print loop would be meaningless and redefinition would probably be undesirable. By making a clear distinction between a programming language and programming environments that support it, we treat the language itself as an abstraction. Such language abstraction is important, for it allows the same language to be used in many different environments.

1.1.3 Conditional Evaluation

We have seen that Scheme definitions cannot be expressed with an application, so a special form must be used. Conditional expressions are a second situation in which a special form is required. The basic conditional expression in Scheme has this syntax:

(if *test-exp then-exp else-exp*)

The expression *test-exp* is evaluated first. If its value is true, *then-exp* is evaluated, and its value is returned as the value of the entire `if` expression. If the value of *test-exp* is false, *else-exp* is evaluated to obtain the value of the `if` expression.

```
> (if #t 1 2)
1
> (zero? 5)
#f
> (if (zero? 5) 1 (+ 1 2))
3
> (define true #t)
> (define false #f)
> (if (zero? 0)
      (if false 1 2)
      3)
2
> (if (if true false true) 2 3)
3
```

The special form `if` cannot be implemented as a procedure. For one thing, only one of *then-exp* or *else-exp* should be evaluated, and it would be inefficient to evaluate both; but there is an even more compelling reason. An important

use of conditionals is to prevent an expression from being evaluated when it is unsafe to do so. For example, we might write

```
(if (zero? a) 0 (/ x a))
```

to make sure that a is nonzero before dividing. In this situation, we say the test *guards* the division. Were `if` a procedure, its arguments would be evaluated before being applied, so the division-by-zero we were trying to avoid would be performed before it could be stopped.

Several other special forms will be introduced later as they are needed, but `define` and `if` are enough to get us started.

1.2 Data Types

In this section we explore some of the data types in Scheme. Scheme implementations vary somewhat in the range of data types they support, and the repertoire of operations on the data types also varies. We discuss only those data types and operations that are required in this book. They should be part of every implementation.

For each data type, we shall be concerned with three things:

1. The set of values of that type.
2. The procedures that operate on that type.
3. The representation of values of that type when they appear internally as literals in programs or externally as characters that are read or printed.

For example, in mathematics the data type of sets consists of the sets themselves, the well-defined operations on these sets (such as union, intersection, and set-difference), and the notation used to represent sets.

It is an error to pass a standard procedure a value that is not of the expected type. For example, it does not make sense to try to add a number to `#t`. *Type checking* is required to detect such *type errors*. If these checks are performed at run time when standard procedures are invoked, as is generally the case for Scheme implementations, we have *dynamic type checking*. In many languages, an analysis is performed at compile time to detect potential type errors. This analysis, which must be based only on the text of the program and not run-time values, is called *static type checking*. It has the advantage of catching errors earlier but requires more complicated and restrictive rules for determining if a program is correctly typed.

1.2.1 Numbers, Booleans, Characters, Strings, and Symbols

We have already used two data types: *number* and *boolean*. Numbers may be included in Scheme programs in the usual way. The operations on numbers include the standard arithmetic operations, such as +, -, *, and /. The type predicate *number?* takes an arbitrary value and returns true if its argument is a number and false otherwise. The equality predicate for numbers is =.

The boolean data type has only two values, true and false, represented by `#t` and `#f`, respectively. Booleans are used primarily in conditional expressions. The type predicate *boolean?* tests an arbitrary value to see if it is a boolean, boolean values may be compared for equality using the predicate *eq?*, and the standard procedure *not* performs logical negation.

```
> (eq? (boolean? #f) (not #f))
#t
```

Characters that are visible when they print are represented as literals by preceding them with `#\`, for example `#\a` and `#\%`. Some nonprinting characters also have literal representations, such as `#\space` and `#\newline`. The character type, equality, and order predicates are *char?*, *char=?*, and *char<?*, respectively, and *char->integer* takes a character and returns an integer representation of the character. The predicates *char-alphabetic?*, *char-numeric?*, and *char-whitespace?* are used to determine the class of a character. The predicate *char-whitespace?* returns true when its argument is a space, return or linefeed character.

```
> (char? #\$)
#t
> (char=? #\newline #\space)
#f
> (char<? #\a #\b)
#t
```

Strings are sequences of characters that are represented by surrounding the characters with double quote marks. The string type predicate is *string?*. The procedure *string-length* takes a string and returns an integer indicating the number of characters in the string. The procedure *string-append* concatenates its arguments to form a new string. The procedures *string->symbol*, *string->number*, and *string->list* convert a string into a symbol, number, and list of characters, respectively. (Symbols and lists will be discussed soon.) The procedure *string* takes any number of arguments, which must be characters, and returns a string of these characters. The procedure *string-ref* takes a string and a nonnegative integer less than the length of the string and returns the character indexed by the integer. Indexing is *zero based,* meaning that the characters are numbered starting with zero.

```
> (define s "This is a.")
> (define ss (string-append s "longer string"))
> (string? s)
#t
> (string-length s)
10
> (string-length ss)
23
> (string-ref s 2)
#\i
> (string #\a #\b)
"ab"
> (string->symbol "abc")
abc
> (string->list s)
(#\T #\h #\i #\s #\space #\i #\s #\space #\a #\.)
```

Programs that process other programs, such as many in this book, frequently manipulate identifiers. Identifiers are central to a number of other kinds of programming, such as artificial intelligence and database applications. In fact, identifiers play an important role in most programs that are not primarily concerned with manipulating numbers. When identifiers are treated as values in Scheme, they are called *symbols*. The manipulation of symbols is greatly facilitated by providing a distinct data type for them.

Just as strings must be surrounded with quote marks to distinguish them from the rest of a program, symbolic literals must be specially marked, for otherwise they would be indistinguishable from variable references. Thus another special form is needed to introduce symbols into programs:

(quote *datum*)

Here *datum* may be a symbol or any other standard external (printed) representation for Scheme data. The value of a quoted literal expression is the associated data value.

```
> (define x 3)
> x
3
> (quote x)
x
> 99
99
> (quote 99)
99
```

Such expressions are used so often that there is an abbreviation for them. The form (quote *datum*) may also be written

'datum

utilizing the single-quote character. Most languages have quoting mechanisms of some sort to avoid confusion between literals and other program elements. The only literals that are "self-quoting," meaning that they may be used directly as expressions without being enclosed in a quote expression, are numbers, booleans, strings, and characters.

Two basic operations on symbols are the symbol type predicate, *symbol?*, and the predicate for testing equality of two symbols, *eq?*,

```
> (define x 3)
> (number? x)
#t
> (symbol? x)
#f
> (number? 'x)
#f
> (symbol? 'x)
#t
> (eq? 'x 'x)
#t
> (eq? 'x 'y)
#f
> (define y 'apple)
> y
apple
> (eq? y (quote apple))
#t
> (eq? y 'y)
#f
```

1.2.2 Lists

A *list* is an ordered sequence of *elements,* which may be of arbitrary types. Lists are a flexible way of combining multiple values into a single *compound* object. Scheme provides convenient facilities for creating and manipulating lists. These facilities, along with most other Scheme data types, are derived from the much older language LISP. (The name stands for LISt Processing.)

A list is represented by surrounding representations of its elements with a pair of parentheses. For example, (a 3 #t) represents a list consisting of three elements: the symbol a, the number three, and the value true. Here are a few more lists

()	the *empty list*
(a)	a list of length 1
((b c d))	a list of length 1 that contains a list of length 3

Just as quote is necessary to distinguish between symbolic literals and variables, it is also necessary to avoid confusing literal lists with procedure calls or special forms. The expression (quote (a b c)) yields the list (a b c) as its value. However, the expression (a b c) is a procedure call whose value depends on the values of the variables a, b, and c; or perhaps it is a special form where a is a keyword.

There are several standard procedures that build new lists. Here we consider the most important ones, *list* and *cons*. The standard procedure *list* may be applied to any number of arguments. It forms a list of their values. (Most procedures take a fixed number of arguments, but *list*, *string* and *string-append* are exceptions.)

```
> (list 1 2 3)
(1 2 3)
> (define x 3)
> (define y 'apple)
> (list x y)
(3 apple)
> (define list-1 '())
> (define list-2 '(a))
> (define list-3 '((b)))
> (list list-1 list-2 list-3 '(((c))))
(() (a) ((b)) (((c))))
> (list)
()
```

The second important list-building procedure, *cons*, always takes two arguments. The first may be any Scheme value, and the second must (for the moment) be a list. If its first argument is the value v and its second argument is the list $(v_0\ v_1\ \ldots\ v_{n-1})$, then *cons* returns the list $(v\ v_0\ v_1\ \ldots\ v_{n-1})$. The returned list is always one longer than the second argument. The name cons stands for *cons*truct, because *cons* constructs a new compound object. (Actually *cons*'s second argument may be anything. But if it is not a list, the value returned will not be a list either. We discuss this further in section 1.2.3.) Study the following examples carefully; they illustrate several important features of *cons*.

```
> (cons 'a '(c d))
(a c d)
> (list 'a '(c d))
(a (c d))
> (cons '(a b) '(c d))
((a b) c d)
> (cons '() '(c d))
(() c d)
> (cons 'a '())
(a)
> (cons '(a b) '())
((a b))
> (define y 'apple)
> (cons y list-2)
(apple a)
> (define list-4 (cons list-1 list-2))
> list-4
(() a)
> (cons list-4 list-3)
((() a) (b))
```

Observe in these examples that if the first element to *cons* is a list, that list becomes an element of the value returned by *cons*. To add all elements of a list to the front of another list (in the same order), the procedure *append* should be used.

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

Compare these results with those obtained by passing the same arguments to *cons*.

The simplest way to divide a list is between the first element and the rest of the list. For historical reasons, the first element of a list is known as its *car* and the rest of the list is known as its *cdr*. The standard procedures *car* and *cdr* select these components of a list. Thus if l is the list $(v_0\ v_1\ \ldots\ v_{n-1})$, then $(car\ l) = v_0$ and $(cdr\ l) = (v_1\ \ldots\ v_{n-1})$. It is an error to call *car* or *cdr* with the empty list.

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

Clearly *car* and *cdr* undo what *cons* does. The exact relationship between *car*, *cdr*, and *cons* is expressed by the equations

$$(car\ (cons\ v\ l)) = v$$
$$(cdr\ (cons\ v\ l)) = l$$

where v is any value, l is any list, and = indicates identical values.

Nested calls to *car* and *cdr* are so common that Scheme provides an assortment of procedures that take care of the more frequent cases. For example, the procedures *cadr* and *caddr* are defined such that

$$(cadr\ l) = (car\ (cdr\ l))$$
$$(caddr\ l) = (car\ (cdr\ (cdr\ l)))$$

The sequence of as and ds surrounded by c and r in the procedure name determines the cars and cdrs and their ordering. The rightmost a/d (car/cdr) is performed first, just as the innermost procedure call is done first.

```
> (cadr '(a b c))
b
> (cddr '(a b c))
(c)
> (caddr '(a b c))
c
```

Empty lists are always represented by the same object, called the *empty list*. (For historical reasons, it is sometimes called the *null* object.) The predicate *null?* tests if its argument is the empty list.

```
> (null? '())
#t
> (define list-2 (list 'a))
> list-2
(a)
> (null? list-2)
#f
> (null? (cdr list-2))
#t
```

∘ *Exercise 1.2.1*

Fill in the blank lines of the following transcript.

```
> (define x '(a b ((3) c) d))
> (car (cdr x))

> (caddr x)

> (cdaddr x)

> (char? (car '(#\a #\b)))

> (cons 'x x)

> (cons (list 1 2) (cons 3 '(4)))

> (cons (list) (list 1 (cons 2 '())))
```

□

1.2.3 Pairs

Most of the time it is desirable to view lists abstractly as we have just done, however, it is sometimes necessary to understand how lists are constructed.

In Scheme, nonempty lists are represented as *pairs*. A pair (sometimes called a *dotted pair* or *cons cell*) is a structure with two fields, called *car* and *cdr*. The procedure *cons* creates a new pair with the car and cdr fields

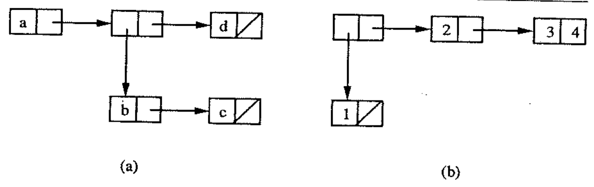

Figure 1.2.1 Box diagrams

initialized to the values of its first and second arguments, respectively. The procedures *car* and *cdr* access the two fields. This explains the behavior introduced in the last section. The type predicate for recognizing pairs is `pair?`.

The structure of values built from pairs is conveniently illustrated by diagrams in which pairs are represented by boxes. Each of these boxes has a left and a right half, representing the car and cdr fields, respectively. Each half contains a pointer to another box if the value of the corresponding field is another pair. If the field value is the empty list, this is represented by a slash through the box. Finally, if the field value is a symbol, number, or boolean, its printed representation is written in the corresponding half of the box. The list `(a (b c) d)` is represented in figure 1.2.1 (a).

If a list has length n, the result of taking the cdr of the list n times must be the empty list. Thus a list is represented by either the empty list or a chain of pairs, linked by their cdr fields, with the empty list in the cdr field of the last pair of the chain. A cdr-linked chain of cons cells that does not end in the empty list is called an *improper list*, even though it is not a list at all. Figure 1.2.1 (b) illustrates such a data structure. We can denote such data structures in a linear format by writing `(a . d)` for a pair whose car is a and whose cdr is d. (Hence the term *dotted pair*.) The data structures in figure 1.2.1 (a) and (b) might be written as

```
(a . ((b . (c . ())) . (d . ())))
```

and

```
((1 . ()) . (2 . (3 . 4)))
```

respectively. Either of these might appear quoted in a Scheme program. This *dot notation* may be intermixed with conventional list notation, so the second structure might also be written

```
((1) 2 3 . 4)
```

Dot notation is required only when writing improper lists.

The predicate *eq?* may be used to compare pairs as well as symbols. In fact, *eq?* may be used to test if any two objects are the same object. The behavior of *eq?* on symbols, booleans, characters, and the empty list is straightforward: if they have the same written representation, they are the same object. This is not necessarily true for numbers, pairs, and strings. The behavior of *eq?* on numbers is implementation dependent. If *eq?* is presented with two pairs (or strings), it returns true if and only if they are the *same* pair (or string). Since *cons* creates a new pair every time it is called, *eq?* must be used with caution on lists.

```
> (define a (cons 3 '()))
> (define b (cons 3 '()))
> a
(3)
> b
(3)
> (eq? a a)
#t
> (eq? a b)
#f
> (eq? (cons 1 2) (cons 1 2))
#f
> (eq? '() '())
#t
```

In this example a and b are different pairs, even though they both print as (3), so they are not "eq to each other." However, every reference to the variable a returns the same pair, so `(eq? a a)` is true.

Pairs may be *shared*. That is, the same pair may be referred to by different variable bindings and pair fields.

```
> b
(3)
> (define c b)
> (eq? b c)
```

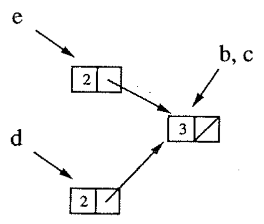

Figure 1.2.2 Box diagram with sharing

```
#t
> (define d (cons 2 c))
> (define e (cons 2 c))
> d
(2 3)
> e
(2 3)
> (eq? d e)
#f
> (eq? (cdr d) (cdr e))
#t
```

Here *b*, *c*, the cdr of *d*, and the cdr of *e* are all the same pair, though *d* and *e* are different pairs. Standard printed notation does not represent sharing, but box diagrams do, as figure 1.2.2 illustrates. The sharing of literals is not specified; for example, `(eq? '(3) '(3))` could be true or false.

There are procedures for assigning new values to the car and cdr fields of an existing pair, which will be discussed in section 4.5. When a pair is modified by one of these procedures, the change is noted in all data structures that share the pair.

The only other way to detect sharing of pairs is by using *eq?*.

• *Exercise 1.2.2*
Fill in the blank lines of the following transcript.

```
> (define x1 '(a b))
> (define x2 '(a))
```

```
> (define x3 (cons x1 x2))
> x1

> (eq? x3 (cons x1 x2))

> (eq? (cdr x3) x2)

> (eq? (car x1) (car x2))

> (cons (cons 'a 'b) (cons 'c '()))

> (cons 1 (cons 2 3))
```

□

1.2.4 Vectors

So far we have seen one means for building compound data objects in Scheme: the cons cell. These cells may be used to construct lists of arbitrary length. Lists are a *derived* data type because they are built using *primitive* data types: the cons cell and the empty list. The advantage of lists is the ease with which new lists may be formed by adding elements to the front of existing lists. However, lists do not provide names for all their elements or random access to them. Compositions of invocations of *car* and *cdr*, also called car/cdr chains, are an awkward way of referring to the first few list elements. It is possible to access list elements via an index number, but with conventional lists access time increases linearly with the index, since to reach a given element it is necessary to traverse the cdr pointers of all the elements that appear earlier in the list.

Neither cons cells nor lists correspond to the two ways to build compound data objects that are most commonly provided by programming languages: records and arrays. Record elements are selected by *field names.* Records are also *heterogeneous,* meaning that their elements may differ in their type. Arrays, on the other hand, are *homogeneous,* in the sense that each of their components must be of the same type, and array components are selected by an index number (or multiple index numbers in the case of multidimensional arrays). Both records and arrays provide *random access* to their components; that is, each component may be accessed in the same amount of time.

Scheme does not provide arrays or records directly. Instead it supplies *vectors,* which may be used in place of arrays and records. Vectors provide random access via index numbers (like arrays) and may be heterogeneous (like records).

The standard procedure *vector* takes an arbitrary number of arguments, such as *list* and *string*, and constructs a vector whose elements contain the argument values. Vectors are written like lists, but with a hash (#) immediately preceding the left parenthesis. By convention, vectors must also be quoted when they appear in programs as literals.

```
> (define v1 (vector 1 2 (+ 1 2)))
> v1
#(1 2 3)
> (define v2 (quote #(a b)))
> v2
#(a b)
> (vector v1 v2)
#(#(1 2 3) #(a b))
> '#(#(a nested vector) (and a list) within a quoted vector)
#(#(a nested vector) (and a list) within a quoted vector)
```

The number of elements in a vector is its *length,* which may be determined with the standard procedure *vector-length*. The type predicate for vectors is *vector?*. The selector *vector-ref* takes a vector and a zero-based index and returns the value of the element indicated by the index. Thus the indices for a given vector are natural numbers in the range from zero through one less than the length of the vector. There are procedures, *vector->list* and *list->vector*, for transforming one compound data type into the other.

```
> (define v3 '#(first second last))
> (vector? v3)
#t
> (vector-ref v3 0)
first
> (vector-length v3)
3
> (vector-ref v3 (- (vector-length v3) 1))
last
> (vector-ref '#(another #((heterogeneous) "vector")) 1)
#((heterogeneous) "vector")
> (vector->list v3)
(first second last)
```

Of course it is an error to pass *vector-ref* an index number that is not a valid index for the given vector.

We use a data structure called a *cell*, a "one-element" vector. In addition to the procedure *make-cell*, which constructs a cell, there is a procedure for referencing a cell, *cell-ref* and one that determines if its argument is a cell, *cell?*. Cells will be useful in chapter 5 for characterizing languages with side effects and in chapter 6 for describing various parameter-passing mechanisms. See exercise 1.3.2 for an implementation of cells.

The procedure *eq?* may again be used to test whether two objects are the *same* vector. An assignment operation for changing the value of a vector element is introduced in section 4.5. As was the case with pairs, the sharing of vectors in data structures may be revealed using *eq?* or assignment.

∘ *Exercise 1.2.3*
Fill in the blank lines of the following transcript

```
> (define v1 (vector (cons 1 2) 3))
> (define v2 (vector 'a v1))
> v2

> (define v3 '#(a #((1 . 2) 3)))
> (eq? v1 v3)

> (eq? v1 (vector-ref v2 1))

> (eq? (vector-ref v1 0)
       (vector-ref (vector-ref v2 1) 0))
```

□

1.3 Procedures

As you might expect, *procedure?* is a Scheme type predicate for procedures. The following transcript illustrates this as well as demonstrating that procedures can be treated as values.

```
> (procedure? 'car)
#f
> (procedure? car)
#t
```

```
> (procedure? (car (list cdr)))
#t
```

The first two examples distinguish between the symbol car and the procedure *car*. The second and third illustrate passing a procedure as an argument to another procedure. In the third, the procedure *cdr* is also stored in a data structure and returned as the value of another procedure, *car*. Here are some more complicated examples

```
> (if (procedure? 3) car cdr)
#<Procedure>
> ((if (procedure? 3) car cdr) '(x y z))
(y z)
> (((if (procedure? procedure?) cdr car)
    (cons car cdr))
   '(car in the car))
(in the car)
> (((if (procedure? procedure?) car cdr)
    (cons car cdr))
   '(x y z))
x
```

Procedures are normally called using the application form, as in (+ 1 2), but sometimes we need to call a procedure with argument values that have already been assembled into a list. The standard procedure *apply* is provided for this purpose. It takes a procedure and a list and returns the result of calling the procedure with the values given in the list.

```
> (apply + '(1 2))
3
> (define abc '(a b c))
> (apply cons (cdr abc))
(b . c)
> (apply apply (list procedure? (list apply)))
#t
```

1.3.1 lambda

We have seen that Scheme supplies a number of standard procedures. It is also possible for the user to create new procedures, which may not be bound to variables. The special form for creating new procedures is lambda. Its most common syntax is

```
(lambda formals body)
```

Here *formals* is a (possibly empty) list of variables, and *body* is any expression. The listed variables are said to be *formal parameters,* or *bound variables,* of the procedure. In many languages, type information must be provided for formal parameters. However, Scheme automatically keeps track of types at run time, so type declarations are not required. (This is more flexible and simplifies code, but has the disadvantage that type errors are not detected until run time, increasing run-time overhead. We shall have more to say about types in chapter 3.) When the procedure is called, the formal parameters (if any) are first *bound to* (associated with) the arguments, and then the body is evaluated. Within the body, the argument values may be obtained by variables that correspond to the formal parameters. Lambda bindings are not accessible outside the body of the procedure: they are said to be *local* to the procedure's body.

For example, a procedure that adds two to its argument may be created by evaluating the expression:

```
(lambda (n) (+ n 2))
```

This expression does not give the procedure a name. Naming is accomplished by another expression, such as a define expression, if desired, however, a procedure may be applied immediately, passed as an argument, or stored in a data structure without ever being named.

```
> ((lambda (n) (+ n 2)) 4)
6
> (list (lambda (n) (+ n 2)) 6)
(#<Procedure> 6)
> (define add2 (lambda (n) (+ n 2)))
> (add2 6)
8
> (define select
    (lambda (b lst)
      (if b
          (car lst)
          (cadr lst))))
> (select #f '(a b))
b
> ((select #t (list cdr car))
   '(a b c))
(b c)
```

Procedures without names, which are not the binding of a variable, are said to be *anonymous.* In most other languages, procedures are never anonymous: they may be created only via declarations that name them. (Of course anonymity is relative to context: if an anonymous procedure is bound to a parameter via procedure call, it is not anonymous in the context of the called procedure.)

Anonymous procedures are often used as arguments. We illustrate this using the procedures *map* and *andmap*, which generally take two arguments: a procedure and a list. The list may be of any length, and the procedure must take one argument. The procedure *map* builds a new list whose elements are obtained by calling the procedure with the elements of the original list. The procedure *andmap* applies the procedure to each element of the list and returns true if all are true. Otherwise it returns false.

```
> (map (lambda (n) (+ n 2)) '(1 2 3 4 5))
(3 4 5 6 7)
> (define add2
    (lambda (n)
      (+ n 2)))
> (map add2 '(1 2 3 4 5))
(3 4 5 6 7)
> (andmap number? '(1 2 3 4 5))
#t
> (map null? '((a) () () (3)))
(#f #t #t #f)
> (andmap null? '((a) () () (3)))
#f
> (map car '((a b) (c d) (e f)))
(a c e)
> (map list '(a b c d))
((a) (b) (c) (d))
> (map (lambda (f) (f '(a b c d)))
       (list car cdr cadr cddr caddr))
(a (b c d) b (c d) c)
```

1.3.2 First-Class Procedures

A value is said to be *first class* if it may be passed to and returned from procedures and stored in data structures. In Scheme, *all* values are first class, including procedures. In other languages, simple values such as numbers are first class, compound values such as records and arrays are sometimes

first class, and procedures are almost never first class. Though it is usually possible to pass procedures as arguments, it is often impossible to return them as values or store them in data structures. (See chapter 10 for a discussion of the implementation of such languages.) First class procedures contribute greatly to the expressive power of a language.

For an example of a procedure that takes procedural arguments and returns a procedural result, consider the problem of defining a procedure that performs functional composition. Assume that f and g are two functions of one argument such that Range(g) $\subseteq$ Domain(f). Then the *composition* of f and g, $f \circ g$, is defined by this equation:

$$(f \circ g)(x) = f(g(x)).$$

The assumption about the range of g and the domain of f ensures that every possible result from g is a possible argument to f. It is straightforward to define composition in Scheme.

```
> (define compose
    (lambda (f g)
      (lambda (x)
        (f (g x)))))
> (define add4 (compose add2 add2))
> (add4 5)
9
> ((compose car cdr) '(a b c d))
b
> ((compose list (compose cdr cdr))
   '(a b c d))
((c d))
```

• *Exercise 1.3.1*
What is unusual about the following expression?

```
((lambda (x)
   (list x (list (quote quote) x)))
 (quote (lambda (x)
          (list x (list (quote quote) x)))))
```

Try to figure out what it does without typing it into a Scheme system. Can similar behavior be achieved without using list? □

• *Exercise 1.3.2*
Here is an implementation of cells.

```
(define cell-tag "cell")

(define make-cell
  (lambda (x)
    (vector cell-tag x)))

(define cell?
  (lambda (x)
    (if (vector? x)
        (if (= (vector-length x) 2)
            (eq? (vector-ref x 0) cell-tag)
            #f)
        #f)))

(define cell-ref
  (lambda (x)
    (if (cell? x)
        (vector-ref x 1)
        (error "Invalid argument to cell-ref:" x))))
```

Fill in the values of the following transcript.

```
> (define c (make-cell 5))
> c

> (cell? c)

> (cell-ref c)
```

□

∘ *Exercise 1.3.3*
Consider two or three other languages you know or for which you can find documentation. What restrictions, if any, are imposed on procedures that keep them from being first class? Is it possible to create anonymous procedures? □

Here is an example of a procedure that takes a numeric value and returns a procedure.

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

When *f* is passed a number x it returns a procedure that takes a number and adds x to it.

```
> (define new-add2 (f 2))
> (new-add2 4)
6
```

Here *new-add2* has the same behavior as the *add2* procedure defined earlier. So what is the point of defining *f*? If a computation requires generating many of these add-n procedures for different values of n, or if the values of n are unknown at the time the program is written, then a procedure like *f* is called for.

```
> (define add3 (f (+ 1 2)))
> (add3 5)
8
> ((f 5) 6)
11
> (define g
    (lambda (a)
      (lambda (d)
        (cons a d))))
> ((g 'a) '(b c))
(a b c)
> (map (g 'a) '((b c) (1 2)))
((a b c) (a 1 2))
```

Having functions of more than one argument is certainly convenient, but it is not absolutely necessary. Using the technique just illustrated, any procedure p of $n \geq 2$ arguments can always be transformed into a procedure p' of one argument that returns a procedure of $n-1$ arguments such that

$$((p'\,x_1)\,x_2 \ldots x_n) = (p\,x_1 \ldots x_n)$$

By repeating this transformation $n-1$ times, we obtain a procedure p'' such that

$$(\ldots((p''\,x_1)\,x_2) \ldots x_n) = (p\,x_1\,x_2 \ldots x_n)$$

This transformation is known as *currying*. The procedures *f* and *g* above are curried versions of the addition and *cons* procedures, respectively. Of course if an existing procedure is to be replaced by a curried version, all calls to the procedure must be changed.

A curried version of a procedure normally takes the first argument first, but this is not always what is desired. The following example illustrates the use of a "reverse-curried" version of *cons*.

```
> (define h
    (lambda (d)
      (lambda (a)
        (cons a d))))
> ((h '(b c)) 'a)
(a b c)
> (map (h '(b c)) '(a 1 2))
((a b c) (1 b c) (2 b c))
```

• *Exercise 1.3.4*
Write a procedure *curry2* that takes a procedure of two arguments and returns a curried version of the procedure that takes the first argument and returns a procedure that takes the second argument. For example,

```
> (((curry2 +) 1) 2)
3
> (define consa ((curry2 cons) 'a))
> (consa '(b))
(a b)
```

□

∘ *Exercise 1.3.5*
Write a curried version of *compose*. Can you think of a use for it? □

∘ *Exercise 1.3.6*
A language could be designed like ML, so that if a procedure is passed fewer arguments than it expects, it simply returns a procedure that takes the rest of the arguments. Thus procedures are "automatically" curried. What are the advantages and disadvantages of this feature? □

1.3.3 Variable Arity Procedures

The *arity* of a procedure is the number of arguments that it takes. Most procedures, including those that result from evaluating lambda expressions of the form introduced so far, have fixed arity. An error message results if a fixed arity procedure is invoked with the wrong number of arguments. Examples of procedures that can take a variable number of arguments are the standard procedures `list`, *vector*, and *string*. It is occasionally necessary to define new procedures of variable arity. This is accomplished with a lambda expression of the form

```
(lambda formal body)
```

where *formal* is a single variable. When the resulting procedure is invoked, this variable is bound to a list of the argument values. The simplest example is `(lambda x x)`, which is equivalent to `list`. A more interesting example is the following procedure, which may be invoked with two or more arguments, in which case it behaves like +, or with one argument, in which case it behaves like a curried +.

```
> (define plus
    (lambda x
      (if (null? (cdr x))
          (lambda (y) (+ (car x) y))
          (apply + x))))
> (plus 1 2)
3
> ((plus 1) 2)
3
```

• *Exercise 1.3.7*
Define a version of compose that takes as arguments either two or three procedures (of one argument) and composes them. The composition of three procedures is specified by this equation:

```
(compose f g h) ⇒ (compose f (compose g h))
```

□

3 Syntactic Abstraction and Data Abstraction

In this chapter we present several special forms that are precisely equivalent to syntactic patterns that are expressible in terms of existing forms. They are examples of *syntactic abstraction,* since they abstract common syntactic patterns. They are informally known as *syntactic sugar,* since they make a language more pleasant to use but add nothing of substance. First we introduce syntactic abstractions that are useful for creating local bindings and performing multi-way branches. We then present a record facility for Scheme that may be implemented via two more syntactic abstractions.

Our implementation of records hides details of how records are implemented. This provides an example of *data abstraction* and leads to a general discussion of *abstract data types.* These data types allow the development of programs that are independent of how their data is represented. One of the benefits of such *representation independence* is that data that is first represented, for simplicity, as first-class procedures may later be represented, for efficiency, as records or other more specialized data representations. The last section illustrates such transformations of data representation, which are extensively used throughout the rest of this book.

3.1 Local Binding

So far we have seen two ways to create bindings in Scheme. Definitions, as we use them, create *top-level* bindings whose region is the entire program. Lambda expressions, which yield procedures, create local bindings for their parameters when invoked. The region of these bindings is restricted to the body of the procedure. There is frequently a need to create local bindings for immediate use, rather than for use when a procedure is invoked. This section introduces two special forms for creating such bindings.

3.1.1 let

Consider the `if` expression of the first *subst* definition in section 2.2.2.

```
(define subst
  (lambda (new old slst)
    ...
    (if (symbol? (car slst))
        (if (eq? (car slst) old)
            (cons new (subst new old (cdr slst)))
            (cons (car slst) (subst new old (cdr slst))))
        (cons (subst new old (car slst))
              (subst new old (cdr slst))))
    ... ))
```

The expression `(subst new old (cdr slst))` appears three times, and its value is always needed if *slst* is not null. It would be clearer if this value could be computed and bound to a variable, say `cdr-result`, before the expression is evaluated and then simply referred to by this name. Since this binding has no significance outside of this expression, the binding should be local to the expression. One way to accomplish this is to use a lambda expression whose body is the expression and invoke the resulting procedure immediately.

```
((lambda (cdr-result)
   (if (symbol? (car slst))
       (if (eq? (car slst) old)
           (cons new cdr-result)
           (cons (car slst) cdr-result))
       (cons (subst new old (car slst))
             cdr-result)))
 (subst new old (cdr slst)))
```

We treat `(car slst)` similarly, since it appears four times.

```
((lambda (car-value cdr-result)
   (if (symbol? car-value)
       (if (eq? car-value old)
           (cons new cdr-result)
           (cons car-value cdr-result))
       (cons (subst new old car-value)
             cdr-result)))
 (car slst)
 (subst new old (cdr slst)))
```

The original expression has now been simplified considerably. Furthermore, in the original expression `(car slst)` is evaluated two or three times (depending on which `if` branches are taken), whereas in the version above it is evaluated only once. This illustrates another advantage of local bindings: they can reduce the amount of computation. In our example, the amount of computation involved in taking the car of *slst* is small enough that this is of little significance, however, sometimes expressions that appear repeatedly involve a great deal of computation.

You probably sense the problem with the approach taken above: it is difficult for the eye to match the formal parameters with their associated operands. Looking at the expression above, it is not obvious that *cdr-result* is the value of `(subst new old (cdr slst))`. The `let` special form is provided to solve this problem. Using `let`, the definition of *subst* becomes

```
(define subst
  (lambda (new old slst)
    (if (null? slst)
        '()
        (let ((car-value (car slst))
              (cdr-result (subst new old (cdr slst))))
          (if (symbol? car-value)
              (if (eq? car-value old)
                  (cons new cdr-result)
                  (cons car-value cdr-result))
              (cons (subst new old car-value)
                    cdr-result))))))
```

In general, `let` expressions have the form

```
(let ((var₁ exp₁)
      ...
      (varₙ expₙ))
  body)
```

The region associated with the declarations of $var_1, \ldots, var_n$ is *body*. Each of the expressions $exp_1, \ldots, exp_n$ is evaluated, the variables $var_1, \ldots, var_n$ are bound to their values, and finally the expression *body* is evaluated and its value is returned. Thus the above form is precisely equivalent to

```
((lambda (var₁ ... varₙ) body)
 exp₁ ... expₙ)
```

In fact, when some Scheme compilers see a `let` expression, they immediately translate it into this form. (They also typically implement `let` or the equivalent form efficiently by avoiding the creation of a procedure, as in figure 5.3.2.)

Sometimes when two local bindings are required, the value of one of them depends on the value of the other. In this case nested `let` expressions must be used. For example,

```
(let ((x 3))
  (let ((y (+ x 4)))
    (* x y)))
```

is *not* equivalent to

```
(let ((x 3)
      (y (+ x 4)))
  (* x y))
```

since in the latter expression the region of the new declaration of `x` is the body of the `let` expression and does *not* include the expression `(+ x 4)` used to define *y*. The `let` expression is equivalent to this lambda expression:

```
((lambda (x y) (* x y))
 3
 (+ x 4))
```

It is clear that `(+ x 4)` is not in the scope of the formal parameter `x`. Nested `let` expressions may also be used to create bindings for the same variable:

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

Here the inner `let` creates a *hole* in the scope of the outer binding of `x`, but the hole does not include the expression `(* x x)`. Thus the expression has value 18. If this is not clear, try replacing each `let` expression by the equivalent application of a procedure created by `lambda`.

• *Exercise 3.1.1*
What are the values of the following two expressions?

```
(let ((x 5) (y 6) (z 7))
  (let ((x 13) (y (+ x y)) (z x))
    (- (+ x z) y)))

(let ((x 5) (y 6) (z 7))
  (+ (let ((z (+ x z))) (* z (+ z x)))
     (let ((z (* x y))) (+ z (* z (- z y))))))
```
□

∘ *Exercise 3.1.2*
Write `let->application`, which takes a `let` expression (represented as a list) and returns the equivalent expression, also represented as a list: an application of a procedure created by a `lambda` expression. Your solution should not change the body of the `let` expression.

```
> (let->application '(let ((x 4) (y 3))
                        (let ((z 5))
                          (+ x (+ y z)))))
((lambda (x y)
   (let ((z 5))
     (+ x (+ y z))))
 4 3)
```
□

3.1.2 letrec

Frequently it is desirable to bind procedures locally. For example, in section 2.2 we defined the procedure *partial-vector-sum* for use by *vector-sum*. If a procedure is not likely to be of use in other contexts, it is good practice to restrict the scope of its binding to the section of code where it is needed. Thus within the definition of *vector-sum* we would like to use something like

```
(let ((partial-vector-sum
        (lambda (von n)
          (if (zero? n)
              0
              (+ (vector-ref von (- n 1))
                 (partial-vector-sum von (- n 1)))))))
  (partial-vector-sum von (vector-length von)))
```

but this does not work. Recall that the region of a `let` binding is restricted to the body of the `let`. The difficulty is that in *partial-vector-sum* the recursive call `(partial-vector-sum von (- n 1))` is not within the scope of the binding for *partial-vector-sum*. The same difficulty arises whenever there is a need to bind a recursive procedure locally.

Scheme provides the special form `letrec` to make local recursive definitions. The general form follows:

```
(letrec ((var₁ exp₁)
          ...
         (varₙ expₙ))
  body)
```

This is similar to `let`, except that the region of the declarations $var_1, \ldots, var_n$ is the entire `letrec` expression, including the expressions $exp_1, \ldots, exp_n$. Thus $exp_1, \ldots, exp_n$ may define mutually recursive procedures. In most uses of `letrec`, $exp_1, \ldots, exp_n$ are lambda expressions, but this is not required. It is required, however, that no reference be made to $var_1, \ldots, var_n$ during the evaluation of $exp_1, \ldots, exp_n$. For example,

```
(letrec ((x 3)
         (y (+ x 1)))
  y)
```

is illegal. This restriction is necessary because the bindings of $var_1, \ldots, var_n$ cannot have values until $exp_1, \ldots, exp_n$ have been evaluated. The requirement is easily met if these are lambda expressions, because references to variables within the body of a lambda expression are evaluated only when the resulting procedure is invoked, not when the lambda expression is evaluated. Thus

```
(letrec ((x 3)
         (y (lambda () (+ x 1))))
  (y))
```

is legal and evaluates to 4.

Using `letrec`, *vector-sum* may be defined as in figure 3.1.1. We no longer need to pass *von* to *partial-vector-sum*, since *von* does not change and the reference to *von* in the *vector-ref* call is within the scope of `vector-sum`. A simple example of mutual recursion using `letrec` is shown in the program of figure 3.1.2, where the procedures *even?* and *odd?* are defined for nonnegative integers.

There are several advantages to using `letrec`, or `let` for procedures, rather than using `define`:

Figure 3.1.1 The procedure `vector-sum` using `letrec`

```
(define vector-sum
  (lambda (von)
    (letrec ((partial-vector-sum
               (lambda (n)
                 (if (zero? n)
                     0
                     (+ (vector-ref von (- n 1))
                        (partial-vector-sum (- n 1)))))))
      (partial-vector-sum (vector-length von)))))
```

Figure 3.1.2 Example of mutual recursion

```
(letrec ((even? (lambda (n)
                  (if (zero? n)
                      #t
                      (odd? (- n 1)))))
         (odd? (lambda (n)
                 (if (zero? n)
                     #f
                     (even? (- n 1))))))
  (even? 3))
```

1. When studying a procedure call, finding a local definition is easier than finding a global one.
2. The code that could be affected by a modification to a procedure is limited to the scope of a local declaration.
3. Frequently the number of arguments required is reduced when a local declaration is used, because some bindings are provided by the context (as with `von` in the `vector-sum` example above).
4. By reducing the number of global definitions, the chance of a conflict occurring because the same name is used for more than one global definition is reduced. In large programs to which many people contribute code, this last point is very important. We shall have more to say about this in chapter 7.

We have seen that Scheme allows any expression to contain local definitions. Some languages allow local procedure declarations only in certain contexts, such as at the head of procedure declarations or even larger units such as files.

• *Exercise 3.1.3*
Rewrite *subst* using `letrec`. □

∘ *Exercise 3.1.4*
The special forms `let` and `letrec` are both binding forms. Extend the definitions of *occurs free* and *occurs bound* to accommodate `let` and `letrec` expressions. Augment your programs *free-vars* and *bound-vars* to take your new rules into account. □

∘ *Exercise 3.1.5*
Extend the language of exercise 2.3.10 to include `let` and `letrec` expressions. Adapt your program *lexical-address* so that it recognizes `letrec` expressions. □

3.2 Logical Connectives

Most programming languages provide a means for expressing the *conjunction* of expressions with the connective *and*, which forms a logical expression that is true if and only if *all* its subexpressions are true. Similarly, logical *disjunction* may be expressed with the connective *or*, which forms an expression that is true if *any* of its subexpressions are true.

In some languages, *and* and *or* are provided as procedures. In this case, all the subexpressions of a conjunction or disjunction are always evaluated, since operands are evaluated prior to a procedure call. This may be unnecessary. As soon as a false subexpression of a conjunction is found, it is known that the entire expression is false, and when a true subexpression of a disjunction is found, true may be returned immediately. Evaluation of all the subexpressions of a conjunction or disjunction not only may result in wasted computation, but also restricts the way in which they may be used. For example, consider this conjunction:

```
(and (pair? x) (number? (car x)))
```

Since *car* may only be passed a pair, the second subexpression may be evaluated only if it is known that the first is true. Thus it would be an error to write such an expression if `and` were a procedure.

Therefore in many languages, including Scheme, logical conjunction and disjunction are implemented as special forms that evaluate their subexpressions