chap2/chap2.md

chap 2

2 Induction, Recursion, and scope

This chapter introduces the fundamental technique of recursive programming, along with its relation to the mathematical technique of induction. The notion of scope, which plays a primary role in programming language, is also presented. Finally, the material in this chapter will improve your facility with the tools introduced in chapter 1. Section 2.1 and section 2.2 introduce techniques for inductively specifying data structures and show how such specifications may be used to guide the construction of recursive programs. Section 2.3 then introduces the notions of variable binding and scope.

The programming exercises are the heart of this chapter. They provide experience that is essential for mastering the technique of recursive programming upon which the rest of this book is based.

[按]

- induction : 归纳法
- recursion : 递归
- scope : 作用域

本章介绍了递归编程方法，虽然这种方法因为早期计算机硬件容量的限制，而不被提倡。

2.1 Inductively Specified Data

We have already seen a number of data types in Scheme. What is a data type in general? For our purposes, it is enough to say that a data type consists of a set of values along with a set of operations on those values. Every time we decide to represent a certain set of quantities in a particular way, we are defining a new data type: the data type whose values are those representations and whose operations are the procedures that manipulate those entities.

[按]

- 数据类型，数据，对数据的操作

When writing code for the operations on a new data type, it is important to know precisely what values may occur as members of the type. In this section we introduce formal techniques for specifying the set of values that belong to a data type.

[按]

- 定义属于某种数据类型的一组数值

2.1.1 Inductive Specification

Inductive specification is a powerful method of specifying a set of values. To illustrate this method, we use it to describe a certain subset of the natural numbers:

归纳的规则来定义数据类型，产生新的数据类型

Define the set S to be the smallest set of natural numbers satisfying the following two properties:

1. $ 0 \in S$ and
2. Whenever $ x \in S $, then $ x+3 \in S $

[按]
使用逻辑符号, 数学的方法来对数据类型的定义进行严格的证明。

"The smallest set" is the one that is a subset of all the sets satisfying properties 1 and 2.

Let us see if we can describe some partial information about S to arrive at a noninductive specification. We know that 0 is in S, by property 1. Since $ 0 \in S $, by property 2 it must be that $ 3 \in S $. Then since $ 3 \in S $, by property 2 we conclude that $ 6 \in S $, and so on. So we see that ll the multiples of 3 are in S. If we let M denote the set of all multiples of 3, we can restate this conclusion as $ M \subseteq S $. But the set M itself satisfies properties 1 and 2. Since S is a subset of every set that satisfies properties 1 and 2, it must be that $ S \subseteq M $. So we deduce that $ S = M $, the set of multiples of 3. This is plausible: we know all the multiples of 3 must be in S, and anything else is extraneous.

[按]
extraneous, 无关，没有直接联系

This is a typical inductive definition. To specify a set $ S $ by induction, we define it to be the smallest set satisfying two properties of the following form:

1. Some specific values must be in $ S $.
2. If certain values are in $ S $, then certain other values are also in $ S $.

Sticking to this recipe guarantees that $ S $ consists precisely of those values inserted by property 1 and those values included by repeated application of property 2. As stated, this recipe is rather vague. It can be stated more precisely, but that would take us too far afield. Instead, let us see how this proces works on some more examples.

The data type list-of-numbers is the smallest set of values satisfying the two properties:

1. The empty list is a list-of-numbers, and
2. If $ l $ is a list-of-numbers and $ n $ is a number, then the pair $ (n \space . \space l) $ is a list-of-numbers.

From this definition we infer the following:

1. () is a list-of-numbers, because of property 1.
2. $ (14 \space . \space ()\space) $ is a list-of-numbers, because 14 is a number and () is a list-of-numbers.
3. $ (3\space.\space(14\space. \space ())) $ is a list-of-numbers, because 3 is a number and $ (14\space.\space())$ is a list-of-numbers.
4. $(-7\space.\space(3\space.\space(14\space.\space()))) $ is a list-of-numbers, because −7−7 is a number and (3 . (14 . ()))(3 . (14 . ())) is a list-of-numbers.
5. Nothing is a list-of-numbers unless it is built in this fashion.

Converting from dot notation to list notation, we see that ()(), (14)(14), (3 14)(3 14),and (− 7 3 14)(−7 3 14) are all members of list-of-numbers.

Backus-Naur Form and Syntactic Derivations

The previous example was fairly straightforward, but it is easy to imagine how the process of describing more complex data types might become quite cumbersome. To remedy this, a notation has been developed to express the same ideas more concisely: **Backus-Naur Form** or **BNF**. We frequently use it to describe the structure of a data type. BNF is also widely used in specifying the syntax of programming languages.

BNF can be used to inductively define a number of sets at once. These sets are called **syntactic categories**, or sometimes **nonterminals**, and are customarily written with angle brackets around the name of the set: . (When no ambiguity results, we shall sometimes refer informally to syntactic categories without using angle brackets or dashes: "list of numbers.") Each syntactic category is defined by a finite set of **rules**, or **productions**. Each rule asserts that certain values must be in the syntactic category.

The BNF definition of list-of-numbers has two rules that correspond to the two properties.

$$\begin{gathered}\langle list\text{–}of\text{–}numbers\rangle ::= () \\ \langle list\text{–}of\text{–}numbers\rangle ::= (\langle number\rangle \ . \ \langle list\text{–}of\text{–}numbers\rangle)\end{gathered}$$

⟨list–of–numbers⟩⟨list–of–numbers⟩ ::= () ::= (⟨number⟩ . ⟨list–of–numbers⟩)

The first rule says that the empty list is in $ \langle list\text{--}of\text{--}numbers \rangle $, and the second says that if nn is in $ \langle number \rangle $ and ll is in $ \langle list\text{--}of\text{--}numbers \rangle$, then $(n \ . \ l)$(n . l) is in $ \langle list\text{--}of\text{--}numbers \rangle$.

Each rule begins by naming the syntactic category being defined, followed by ::= (read $is$is). The right-hand side of each rule specifies a method for constructing members of the syntactic category in terms of other syntactic categories and $ terminal\space symbols$, such as the left and right parentheses, and the period in the previous example.

Often some syntactic categories mentioned in a BNF rule are left undefined, such as $\langle number\rangle$⟨number⟩. Defining all categories would needlessly complicate the rule if it is safe to assume the reader knows what some of the categories are.

BNF is often extended with a few notational shortcuts. One can write a set of rules for a single syntactic category by writing the left-hand side and ::= just once, followed by all the right-hand sides separated by the special symbol || (vertical bar, read oror). A $\langle list\text{–}of\text{–}numbers\rangle$⟨list–of–numbers⟩ can then be defined by

$$\langle list\text{–}of\text{–}numbers\rangle ::= ()|(\langle number\rangle \ . \ \langle list\text{–}of\text{–}numbers\rangle)$$

⟨list–of–numbers⟩ ::= ()|(⟨number⟩ . ⟨list–of–numbers⟩)

Another shortcut is the $Kleene\ star$Kleene star, expressed by the notation $\{...\}^*$\{...\}∗. When this appears in a right-hand side, it indicates a sequence of any number of instances of whatever appears between the braces. This includes the possibility of no instances at all. Using the Kleene star, the definition of $\langle list\text{–}of\text{–}numbers\rangle$⟨list–of–numbers⟩ in list notation is simply

$$\langle list\text{–}of\text{–}numbers\rangle ::= (\{\langle number\rangle\}^*)$$

⟨list–of–numbers⟩ ::= ({⟨number⟩}∗)

If there are zero instances, we get the empty list. A variant of the star notation is $Kleeneplus\{...\}^+$Kleeneplus{...}+, which indicates a sequence of oneone or more instances. Substituting $^+$+ for *∗ in the above example would define the syntactic category of nonempty lists of numbers. These notational shortcuts are just that –– it is always possible to do without them by using additional BNF rules.

If a type is specified use BNF rules , a $syntactic\ derivation$syntactic derivation may be used to prove that a given data value is a member of the type. Such a derivation starts with the nonterminal corresponding to the type. At each step, indicated by an arrow ⇒⇒, a nonterminal is replace by the right-hand side of a corresponding rule, or with a known member of its syntactic class if the class was left undefined. For example, the previous demonstration that (14 . ())(14 . ()) is a list of numbers may be formalized with the following syntactic derivation:

$$\begin{aligned}&\langle list\text{–}of\text{–}numbers\rangle \\ &\Rightarrow(\langle number\rangle \ . \ \langle list\text{–}of\text{–}numbers\rangle) \\ &\Rightarrow(14 \ . \ \langle list\text{–}of\text{–}numbers\rangle) \\ &\Rightarrow(14 \ . \ ())\end{aligned}$$

⇒⇒⇒⟨list–of–numbers⟩(⟨number⟩ . ⟨list–of–numbers⟩)(14 . ⟨list–of–numbers⟩)(14 . ())

The order in which nonterminals are replaced is not significant. Thus another possible derivation of (14 . ())(14 . ()) is

$$\begin{aligned}&\langle list\text{–}of\text{–}numbers\rangle \\ &\Rightarrow(\langle number\rangle \ . \ \langle list\text{–}of\text{–}numbers\rangle) \\ &\Rightarrow(\langle number\rangle \ .()) \\ &\Rightarrow(14 \ . \ ())\end{aligned}$$

⇒⇒⇒⟨list–of–numbers⟩(⟨number⟩ . ⟨list–of–numbers⟩)(⟨number⟩ .())(14 . ())

◦◦ Exercise 2.1.1

Write a syntactic derivation that proves (− 7 . (3 . (14 . ())))(−7 . (3 . (14 . ()))) is a list of numbers. □□

2.1.3 using BNF to Specify Data2.1.3 using BNF to Specify Data

The term $datum$datum refers to any literal data representation. BNF may be used to specify concisely the syntactic category of data in Scheme. We have seen that numbers, symbols, booleans, and strings all have literal representations, which we associate with the syntactic categories $\langle number\rangle$⟨number⟩, $\langle symbol\rangle$⟨symbol⟩, $\langle boolean\rangle$⟨boolean⟩, and $\langle string\rangle$⟨string⟩, respectively. Section 1.2 informally introduced representations for lists, improper lists (which end with dotted pairs), and vectors. These compound data types contain components that may be numbers, symbols, booleans , strings , or other lists, improper lists or vectors. This is formally specified by the following BNF rules

---

$$\begin{aligned}\langle list\rangle &::= (\{\langle datum\rangle\}^*) \\ \langle dotted\text{–}datum\rangle &::= (\{\langle datum\rangle\}^+ \ . \ \langle datum\rangle) \\ \langle vector\rangle &::= \#(\{\langle datum\rangle\}^*) \\ \langle datum\rangle &::= \langle number\rangle|\langle symbol\rangle|\langle boolean\rangle \\ &\quad |\langle string\rangle|\langle list\rangle|\langle dotted\text{–}datum\rangle|\langle vector\rangle\end{aligned}$$

⟨list⟩⟨dotted–datum⟩⟨vector⟩⟨datum⟩ ::= ({⟨datum⟩}∗) ::= ({⟨datum⟩}+ . ⟨datum⟩) ::= #({⟨datum⟩}∗) ::= ⟨number⟩|⟨symbol⟩|⟨boolean⟩|⟨string⟩|⟨list⟩|⟨dotted–datum⟩|⟨vector⟩

These four syntactic categories are all defined in terms of each other. This is legitimate because there are some simple possibilities for data that are not defined in terms of the other categories.

To illustrate the use of this grammar, consider the following syntactic derivation proving that $(\#t\ (foo\ .\ ())\ 3)$(#t (foo . ()) 3) is a datum.

$$\begin{aligned}&\langle list\rangle \\ &\Rightarrow(\langle datum\rangle\ \langle datum\rangle\ \langle datum\rangle) \\ &\Rightarrow(\langle boolean\rangle\ \langle datum\rangle\ \langle datum\rangle) \\ &\Rightarrow(\#t\ \langle datum\rangle\ \langle datum\rangle) \\ &\Rightarrow(\#t\ \langle dotted\text{–}datum\rangle\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (\{\langle datum\rangle\}^+\ .\ \langle datum\rangle)\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (\langle symbol\rangle\ .\ \langle datum\rangle)\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (foo\ .\ \langle datum\rangle)\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (foo\ .\ \langle list\rangle)\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (foo\ .\ ())\ \langle datum\rangle) \\ &\Rightarrow(\#t\ (foo\ .\ ())\ \langle number\rangle) \\ &\Rightarrow(\#t\ (foo\ .\ ())\ \langle 3\rangle)\end{aligned}$$

⇒⇒⇒⇒⇒⇒⇒⇒⇒⇒⇒⟨list⟩(⟨datum⟩ ⟨datum⟩ ⟨datum⟩)(⟨boolean⟩ ⟨datum⟩ ⟨datum⟩)(#t ⟨datum⟩ ⟨datum⟩)(#t ⟨dotted–datum⟩ ⟨datum⟩)(#t ({⟨datum⟩}+ . ⟨datum⟩) ⟨datum⟩)(#t (⟨symbol⟩ . ⟨datum⟩) ⟨datum⟩)(#t (foo . ⟨datum⟩) ⟨datum⟩)(#t (foo . ⟨list⟩) ⟨datum⟩)(#t (foo . ()) ⟨datum⟩)(#t (foo . ()) ⟨number⟩)(#t (foo . ()) ⟨3⟩)

All three elements of the outer list were introduced at once. This shortcut was possible because the grammar uses a Kleene star. Of course, the Kleene star and plus notation could be eliminated by introducing new nonterminals and productions, and the three list elements would then be introduced with three derivation steps instead of one.

◦◦ Exercise 2.1.2

Rewrite the $\langle datum\rangle$⟨datum⟩ grammar without using the Kleene star or plus. Then indicate the changes to the above derivation that are required by your grammar. □□

◦◦ Exercise 2.1.3

=======

◦◦ Exercise 2.1.2

Rewrite the $\langle datum\rangle$⟨datum⟩ grammar without using the **Kleene star** or **plus**. Then indicate the changes to the above derivation that are required by your grammar. □□

◦◦ Exercise 2.1.3

Write a syntactic derivation that proves $(a\ \text{“}mixed\text{”}\#(bag\ (of\ .\ data)))$(a “mixed”#(bag (of . data))) is a datum , using either the grammar in the book or your grammar from the last exercise. What can you say about $(a\ .b\ .\ c)$(a .b . c) ? □□

[答]

$$\begin{aligned}&(a\ \text{“}mixed\text{”}\ \#(bag\ (of\ .\ data))) \\ &\Rightarrow(\langle datum\rangle\ \langle datum\rangle\ \langle datum\rangle) \\ &\Rightarrow(\langle symbol\rangle\ \langle string\rangle\ \langle vector\rangle) \\ &\Rightarrow(\langle list\rangle)\end{aligned}$$

⇒⇒⇒(a “mixed” #(bag (of . data)))(⟨datum⟩ ⟨datum⟩ ⟨datum⟩)(⟨symbol⟩ ⟨string⟩ ⟨vector⟩)(⟨list⟩)

Let us consider the BNF definitions of some other useful data types. Many symbol manipulation procedures are designed to operate on lists that contain only symbols and other similarly restricted lists. We formalize this notion with their rules:

$$\begin{aligned}\langle s\text{–}list\rangle &::= (\{\langle symbol\text{–}expression\rangle\}^*) \\ \langle symbol\text{–}expression\rangle &::= \langle symbol\rangle|\langle s\text{–}list\rangle\end{aligned}$$

⟨s–list⟩⟨symbol–expression⟩ ::= ({⟨symbol–expression⟩}∗) ::= ⟨symbol⟩|⟨s–list⟩

The literal representation of an s-list contains only parentheses and symbols.

For example

$ (a \space b \space c) $

$ (an \space (((s\text{--}list)) \space (with \space lots) \space ((of) \space nesting)))$

A binary tree with numeric leaves and interior nodes labeled with symbols may be represented using three-element lists for the interior nodes as follows:

$ \langle tree \rangle \space ::= \space \langle number \rangle | (\langle symbol \rangle \space \langle tree \rangle)$

Examples of such trees follow:

1 2
1 2

BNF rules are said to be *context free* because a rule defining a given syntactic category may be applied in any context that makes reference to that syntactic category. Sometimes this is not restrictive enough. For example, a node in a binary search tree is either empty or contains a key and two subtrees.

$$\langle bin\text{-}search\text{-}tree\rangle ::= () | (\langle key\rangle \ \langle bin\text{-}search\text{-}tree\rangle \langle bin\text{-}search\text{-}tree\rangle)$$

⟨bin–search–tree⟩ ::= ()|(⟨key⟩ ⟨bin–search–tree⟩⟨bin–search–tree⟩)

This correctly describes the structure of each node but fails to mention an important fact about binary search tree: all the keys in the left subtree are less than (or equal to) the key in the current node, and all the keys in the right subtree are greater than the key in the current node. Such contraints are said to be **context sensitive**, because they depend on the context in which they are used.

Context-sensitive constraints also arise when specifying the syntax of programming languages. For instance, in many languages every identifier must be declared before it is used. This constraint on the use of identifiers are sensitive to the context of their use. Formal methods can be used to specify context-sensitive constraints, but these methods are far more complicated than BNF. In practice, the usual approach is first to specify a context-free grammar using BNF. Context-sensitive constraints are then added using other methods, usually prose, to complete the specification of a context-sensitive syntax.

Induction

Having described data types inductively, we can use the inductive definitions in two ways: to prove theorems about members of the data type and to write programs that manipulate them. Writing the program is the subject of the next two sections; here we present an example of such a proof.

$Theorem.$Theorem. Let s $\in \langle tree \rangle$. Then s contains an odd number of nodes.

Proof. The proof is by induction on the size of s, where we take the size of s to be the number of nodes in s. The induction hypothesis, $IH(k)$, is that any tree of $size \le k$ has an odd number of nodes. We follow the usual prescription for an inductive proof: we first prove that $IH(0)$ is true, and we then prove that whenever k is a number such that IH is true for k, then IH is true for $k+1$ also.

i. There are no trees with 0 nodes, so $IH(0)$ holds trivially.
ii. Let k be a number such that $IH(k)$ holds, that is, any tree with $\le k$ nodes actually has an odd number of nodes. We need to show that $IH(k+1)$ holds as well: that any tree with $\le k + 1$ nodes has an odd number of nodes. If s has $\le k+1$ nodes, there are exactly two possibilities according to the BNF definition of $\langle tree \rangle$:

a. s could be of the form n, where n is a number. In this case, s has exactly one node, and one is odd.

b. s could be the form $(sym \ s_1 \ s_2)$, where sym is a symbol and s_1 and s_2 are trees. Now s_1 and s_2 must have fewer nodes than s. Since s has $\le k+1$ nodes, s_1 and s_2 must have $\le k$ nodes. Therefore they are covered by $IH(k)$, and they must each have an odd number of nodes, say $2n_1+1$ and $2n_2+1$ nodes, respectively. Therefore the total number of nodes in the tree, counting the two subtrees and the root , is

$$(2n_1 + 1) + (2n_2 + 1) + 1 = 2(n_1 + n_2 + 1) + 1$$

(2n1 + 1) + (2n2 + 1) + 1 = 2(n1 + n2 + 1) + 1

which is once again odd.

This completes the proof of the claim that $IH(k+1)$ holds and therefore completes the induction. □□

The key to the proof is that the substructures of a tree s are always smaller than s itself. Therefore the induction might be rephrased as follows:

1. IH is true on simple structures (those without substructures).
2. If IH is true on the substructures of s, then it is true on s itself.

2.2 Recursively Specified Programs

In the previous section, we used the method of inductive definition to characterize complicated sets. Starting with simple members of the set, the BNF rules were used to build more and more complex member of the set. We now use the same idea to define procedures for manipulating those sets. First we define the simple parts of a procedure's behavior (how it behaves on simple inputs), and then we use this behavior to define more complex behaviors.

Imagine we want to define a procedure to find powers of numbers, $e.g.$ $e(n, x) = x^n$, where n is a nonnegative integer and $x \neq 0$. It is easy to define a sequence of procedures that compute particular powers: $e_0(x) = x^0$, $e_1(x) = x^1$, $e_2(x) = x^2$:

$$\begin{aligned} e_0(x) &= 1 \\ e_1(x) &= x \times e_0(x) \\ e_2(x) &= x \times e_1(x) \\ e_3(x) &= x \times e_2(x) \end{aligned}$$

e0(x)e1(x)e2(x)e3(x) = 1 = x × e0(x) = x × e1(x) = x × e2(x)

In general, if n is a positive integer,

$$e_n(x) = x \times e_{n-1}(x)$$

en(x) = x × en−1(x)

At each stage, we use the fact that the problem has already been solved for smaller n. We are using mathematical induction. Next the subscript can be removed from e by making it a parameter.

1. If n is 0, $e(n, x) = 1$.
2. If n is greater than 0, we assume it is known how to solve the problem for $n-1$. That is , we assume that $e(n-1, x)$ is well defined. Therefore, $e(n, x) = x \times e(n-1, x)$.

To prove that $e(n, x) = x^n$ for any nonnegative integer n, we proceed by induction on n:

1. (Base Step) When $n = 0$, $e(0, x) = 1 = x^0$.
2. (Induction Step) Assume that the procedure works when its first argument is k, that is , $e(k, x) = x^k$ for some nonnegative integer k. Then we claim that $e(k+1, x) = x^{k+1}$. We calculate as follows

$$\begin{aligned} e(k+1, x) &= x \times e(k, x) && \text{(definition of } e) \\ &= x \times x^k && (\text{IH } at \ k) \\ &= x^{k+1} && \text{(fact about exponentiation)} \end{aligned}$$

e(k + 1, x) = x × e(k, x) = x × xk = xk+1 (definition of e)(IH at k)(fact about exponentiation)

This completes the induction.

We can write a program to compute e based upon the inductive definition

```
(define e
    (lambda (n x)
        (if (zero? n)
            1
            (* x
                (e (- n 1) x)))))
```

The two branches of the if expression correspond to the two cases detailed in the definition.

If **we can reduce a problem to a smaller subproblem, we can call the procedure that solved the problem to solve the subproblem.** The solution it returns for the subproblem may then be used to solve the original problem. This works because each time we call the procedure, it is called with a smaller problem, until eventually it is called with a problem that can be solved directly, without another call to itself.

In the above example, we used introduction on integers, so the subproblem was solved by recursively calling the procedure with a smaller value of n. When manipulating inductively defined structures, subproblems are usually solved by calling the procedure recursively on a substructure of the original.

When a procedure calls iteself in this manner , its is said to be **recursively defined**. Such **recursive calls** are possible in Scheme and most other languages. The general phenomenon is known as *recursion*, and it occurs in contexts other than programing, such as inductive definitions. Later we shall study how recursion is implemented in programming languages.

2.2.1 Deriving Programs from BNF Data Specifications

Recursion is a powerful programming technique that is used extensively throughout this book. It requires an approach to programming that differs significantly from the style commonly used in statement-oriented languages. For this reason, we devote the reset of this section to this style of programming.

A BNF definiton for the type of data being manipulated serves as a guide both to where recursive calls should be used and to which base cases need to be handled. This is a funcamental point: *when defining a program based on structural induction, the structure of the program should be patterned after the structure of the data*.

A typical kind of program based on inductively defined structures is a predicate that determines whether a given value is a member of a particular data type. Let us write a Scheme predicate, list-of-numbers? that takes a datum and determines whether it belongs to the syntactic category .

```
> (list-of-numbers? '(1 2 3))
#t
> (list-of_numbers? '(1 two 3))
#f
> (list-of-numbers? '(1 . 2))
#f
```

Recall the definition of :

$$\langle list\text{-}of\text{-}numbers\rangle ::= () | (\langle number\rangle . \langle list\text{-}of\text{-}numbers\rangle)$$

⟨list–of–numbers⟩ ::= ()|(⟨number⟩.⟨list–of–numbers⟩)

We begin by writing down the simplest behavior of the procedure: what it does when the input is the empty list.

```
(define list-of-numbers?
    (lambda (lst)
        (if (null? lst)
            ...
            ...)))
```

By definition, the empty list is a . Otherwise *lst* is not a unless it is a pair.

```
(define list-of-numbers?
    (lambda (lst)
        (if (null? lst)
            #t
            (if (pair? lst)
                ...
                #f))))
```

(Throughout this book, bars in the left margin indicate lines that have changed isnce an earlier version of the same definition.) If *lst* is a pair, there are two alternatives : either the first element is a number, or it is not. If not, the original value cannot be a list of numbers, so we write

```
(define list-of-numbers?
    (lambda (lst)
        (if (null? lst)
            #t
            (if (pair? lst)
                (if (number? (car lst))
                    ...
                    #f)
                #f))))
```

The only case left to consider is when the first element of the list in question passes the *number?* test. According to the definition of , a nonempty list belongs to if and only if its first element is a number and its cdr belongs to . Since we already know that *lst* is nonempty and its car is a number, we can deduce that *lst* is a list of numbers if and only if its cdr is a list of numbers. Therefore we write

```
(define list-of-numbers?
    (lambda (lst)
        (if (null? lst)
            #t
            (if (pair? lst)
                (if (number? (car lst))
                    (list-of-numbers? (cdr lst))
                    #f)
                #f))))
```

To prove the correctness of *list-of-numbers?*, we would like to use induction on the length of *lst*. Howerver, the argument to list-of-numbers? may not be a list at all. This prompts us to define the list-size of a datum to be zero if the datum is not a list and its length if it is a list. We now proceed by induction on the list-size.

1. *list-of-numbers?* works on data of list-size 0, since the only list of lenght 0 is the empty list, for which the corret answer, true, is returned, and if *list-of-numbers?* is not a list, the correct answer, false is returned.
2. Assuming *list-of-numbers?* works on lists of length k, we show that it works on lists of length $k+1$. Let *lst* be such a list. By the definition of , *lst* belongs to if and only if its car is a number and its cdr belongs to . Since *lst* is of length $k+1$, its cdr is of length k, so by the induction hypothesis we can determine the cdr's membership in by passing it to *list-of-numbers?*. Hence *list-of-numbers?* correctly computes membership in for lists of length $k+1$, and the induction is complete.

The recursion terminates because every time *list-of-numbers?* is called, it is passed a shorter list. (This assumes lists are finite, which will always be the case unless the list mutation techniques introduced in section 4.5.3 have been used.)

As a second example, we define a procedure *nth-elt* that takes a list *lst* and a zero-based index *n* and returns element number *n* of *lst*.

```
> (nth-elt '(a b c) 1)
b
```

The procedure *nth-elt* does for lists what *vector-ref* does for vectors. (Actually, Scheme provides the procedure *list-ref*, which is the same as *nth-elt* except for error reporting, but we chose another name because standard procedures should not be tampered with unnecessarily.)

When *n* is 0, the answer is simply the car of *lst*. If *n* is greater than 0, then the anser is element $n-1$ of *lst*'s cdr. Since neither the car nor cdr of *lst* exist if *lst* is the empty list, we must guard the *car* and *cdr* operations so that we do not take the car or cdr of an empty list.

```
(define nth-elt
    (lambda (lst n)
        (if (null? lst)
            (error "nth-elt: list too short")
            (if (zero? n)
                (car lst)
                (nth-elt (cdr lst) (- n 1))))))
```

The procedure *error* signals an error by printing its arguments, in this case a single string, and then aborting the computation. (*error* is not a standard Scheme procedure, but most implementations provide something of the sort. See appendix D and check your Scheme language reference manual for details.) If error checking were omitted, we would have to rely on *car* and *cdr* to complain about being passed the empty list, but their error messages would be less helpful. For example, if you receive an error message from *car* , you might have to look for uses of *car* throughout your program. Even this would not find the error if *nth-elt* were provided by someone else, so that its definition was not a part of your program.

Let us try one more example of this kind before moving on to harder examples. The standard procedure *length* determines the number of elements in a list.

```
> (length '(a b c))
3
> (length '((x) ()))
2
```

We write our own procedure, called *list-length*, to do the same thing. The length of the empty list is 0.

```
(define list-length
    (lambda (lst)
        (if (null? lst)
            0
            ...)))
```

The blank is filled in by observing theat the length of a nonempty list is one more than the length of its cdr.

```
(define list-length
    (lambda (lst)
        (if (null? lst)
            0
            (+ 1 (list-length (cdr lst))))))
```

∘∘ Exercise 2.2.1
The procedures *nth-elt* and *list-length* do not check whether their arguments are of the expected type. What happens on your Scheme system if they are passed symbols when a list is expected? What is the behavior of *list-ref* and *length* in such cases? Write your own versions that guard against these situations. Is it always necessary to signal errors when this occurs, or can a sensible value sometimes be returned? When is it worth the effort to check that arguments are of the right type? Why? □□

2.2.2 Three Important Examples

In this section, we present three simple recursive procedures that will be used as examples later in this book. As in previous examples, they are defined so that (1) the structure of a program reflects the structure of its data and (2) recursive calls are employed at points where recursion is used in the data type's inductive definition.

the first procedure is *remove-list*, which takes two arguments: a symbol, s, and a list of symbols, *los*. It returns a list with the same elements arranged in the same order as *los*, except that the first occurrence of the symbol *s* is removed. If there is no occurrence of *s* in *los*, then *los* is returned.

```
> (remove-first 'a '(a b c))
(b c)
> (remove-first 'b '(e f g))
(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
(c1 c1 a4)
> (remove-first 'x '())
()
```

Before we start on the program, we must complete the problem specification by defining the data type . Unlike the s-lists introduced in the last section, these lists of symbols do not contain sublists.

$$\langle list\text{-}of\text{-}symbols\rangle ::= ()|(\langle symbol\rangle . \langle list\text{-}of\text{-}symbols\rangle)$$

⟨list–of–symbols⟩ ::= ()|(⟨symbol⟩.⟨list–of–symbols⟩)

A list of symbols is either the empty list or a list whose car is a symbol and whose cdr is a list of symbols. If the list is empty, there are no occurrences of *s* to remove, so the answer is the empty list.

```
(define remove-first
    (lambda (s los)
        (if (null? los)
            '()
            ...)))
```

If *los* is nonempty, is there some case where we can determine the answer immediately? If $los = (s\ s_1\ \ldots\ s_{n-1})$los = (s s1 ... sn−1), the first occurrence of *s* is as the first element of *los*. So the result of removing it is just $(s_1\ \ldots\ s_{n-1})$(s1 ... sn−1).

If the first element of *los* is not *s*, say $los = (s_0\ s_1\ \ldots\ s_{n-1})$los = (s0 s1 … sn−1), then we know that s_0s0 is not the first occurrence of ss. Therefore the first element of the answer must be s_0s0. Furthermore, the first occurrrence of ss in loslos must be its first occurrence in $(s_1\ \ldots\ s_{n-1})$(s1 … sn−1). So the rest of the answer must be the result of removing the first occurrence of ss from the cdr of loslos. Since the cdr of loslos is shorter than loslos, we may recursively call $remove\text{–}first$remove–first to remove ss from the cdr of loslos. Thus using $(cons\ (car\ los)\ (remove\text{–}first\ s\ (cdr\ los)))$(cons (car los) (remove–first s (cdr los))), the answer may be obtained. With this, the complete definition of *remove-first* follows.

```
(define remove-first
    (lambda (s los)
        (if (null? los)
            '()
            (if (eq? (car los) s)
                (cdr los)
                (cons (car los) (remove-first s (cdr los)))))))
```

∘∘ Exercise 2.2.2
In the definition of *remove-first*, if the inner *if*'s alternative $(cons\ \ldots)$(cons …) were replaced by $(remove\text{–}first\ s\ (cdr\ los))$(remove–first s (cdr los)), what function would be resulting procedure compute? □□

The second procedure is *remove*, defined over symbols and lists of symbols. It is similar to *remove-first*, but it removes all occurrences of a given symbol from a list of symbols, not just the first.

```
> (remove 'a4 '(c1 a4 d1 a4))
(c1 d1)
```

Since *remove-first* and *remove* work on the same input, their structure is similar. If the list *los* is empty, there are no occurrences to remove, so the answer is again the empty list. If *los* is nonempty, there are again two cases to consider. If the first element of *los* is not *s*, the answer is obatined as in *remove-first*.

```
(define remove
    (lambda (s los)
        (if (null? los)
            '()
            (if (eq? (car los) s)
            ...
            (cons (car los) (remove s (cdr los)))))))
```

If the first element of *los* is the same as *s*, certainly the first element is not to be part of the result. But we are not quite done: all the occurrences of *s* must still be removed from the cdr of *los*. Once again this may be accomplished by invoking *remove* recursively on the cdr of *los*.

```
(define remove
    (lambda (s los)
        (if (null? los)
            '()
            (if (eq? (car los) s)
                (remove s (cdr los))
                (cons (car los) (remove s (cdr los)))))))
```

∘∘ Exercise 2.2.3
In the definition of *remove*, if the inner *if*'s alternative *(cons ...)* were replaced by *(remove s (cdr los))*, what function would the resulting procedure compute? □□

The last of our examples is *subst*. It takes three arguments: two symbols, *new* and *old*, and an s-list *slst*. All elements of *slst* are examined, and a new list is returned that is similar to *slst* but with all occurrences of *old* replaced by instances of *new*.

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

Since *subst* is defined over s-lists, its organization reflects the definition of s-lists

$$\langle s\text{–}list\rangle ::= (\langle symbol\text{–}expression\rangle^*)$$
$$\langle symbol\text{–}expression\rangle ::= \langle symbol\rangle|\langle s\text{–}list\rangle$$

⟨s–list⟩⟨symbol–expression⟩ ::= (⟨symbol–expression⟩∗) ::= ⟨symbol⟩|⟨s–list⟩

If the list is empty, there are no occurrences of *old* to replace.

```
(define subst
    (lambda (new old slst)
        (if (null? slst)
            '()
            ...)))
```

If *slst* is nonempty, its car is a member of and its cdr is another s-list. Thus the program brances on the type of the symbol expression in the car of slst. If it is a symbol, we need to ask whether it is the same as the symbol *old*. If it is , the car of the new answer is *new*; if not, the car of the answer is the same as the car of *slst*. In either case, to obtain the answer's cdr, we need to change all occurrences of *old* to *new* in the cdr of *slst*. Since the cdr of *slst* is a smaller list, we may use recursion.

```
(define subst
    (lambda (new old slst)
        (if (null? 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))))
                ))))
```

In the final case to be considered the car of *slst* is a list. Since the car and cdr of *slst* are both lists, the answer is obtained by invoking *subst* on both and consing the results together.

```
(define subst
    (lambda (new old slst)
        (if (null? 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 olf (cdr slst)))))))
```

This definition has been completed by following the structure of and then and checking for *old* when dealing with symbols.

The subexpression (subst new old (cdr slst)) appears three times in the above definition. This redundancy can be eliminated by noting that when *slst* is nonnull, the answer's car and cdr may be independently computed and then combined with *cons*. The answer's cdr is obtained by invoking *subst* on the cdr of *slst*. The answer's car is obtained by substituting *new* for *old* in the car of *slst*, but the type of *slst*'s car is , not , so *subst* cannot be used directly. The solution is to make a separate procedure for handling substitutions on members of .

```
(define subst
    (lambda (new old slst)
        (if (null? slst)
            '()
            (cons (subst-symbol-expression new old (car slst))
                (subst new old (cdr slst))))))
(define subst-symbol-expression
    (lambda (new old se)
        (if (symbol? se)
            (if (eq? se old) new se)
            (subst new old se))))
```

Since we have strictly followed the BNF definition of and , this recursion is guaranteed to halt. Observe that *subst* and *subst-symbol-expression* call each other recursively. Such procedures are said to be *mutually recursive*.

∘∘ Exercise 2.2.4
In the last line of *subst-symbol-expression*, the recursion is on *se* and not a smaller substructure. Why is the recursion guaranteed to halt? □□

∘∘ Exercise 2.2.5
Write *subst* using *map*. □□

The docomposition of *subst* into two procedures, one for each syntactic category, is an important technique. It allows us to think about one syntactic category at a time, which is important in more complicated situations.

There are many other situations in which it may be helpful or necessary to introduce auxiliary procedures to solve a problem. Always feel free to do so. In some cases the new procedure is necessary in order to introduce an additional parameter. As an example, we consider the problem of summing all the values in a vector.

Since vectors require a program structure that differs from the ones we have used for lists , let us first solve the problem of summing the values in a list of numbers. This problem has a natural recursive solution because nonempty lists decompose naturally into their car and cdr components. We return 0 as the sum of the elements in the empty list.

```
(define list-sim
    (lambda (lon)
        (if (null? lon)
            0
            (+ (car lon)
                (list-sum (cdr lon))))))
```

It is not possible to proceed in this way with vectors, because they do not decompose as readily. Sometimes the best way to solve a problem is to solve a more general problem and use it to solve the original problem as a special case. For the vector sum problem, since we cannot decompose vectors, we generalize the problem to compute the sum of part of the vector. We define *partial-vecotr-sum*, which takes a vector of numbers, *von*, and a number, *n*, and returns the sum of the first *n* values in von.

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

Observe that *von* does not change. In the next chapter we shall see how the conceptual overhead of passing such parameters may be avoided. Since *n* decreases steadily to zero, a proof of correctness for this program would proceed by induction on n. It is now a simple matter to solve our original problem

```
(define vector-sum
    (lambda (von)
        (partial-vector-sum von (vector-length von))))
```

∘∘ Exercise 2.2.6
Prove the correctness of *partial-vector-sum* with the following assumptions: $0 \le n < length(von)$0 ≤ n < length(von). □□

Getting the knack of writing recursive programs involves practice. Thus we conclude this section with a number of exercises.

•• Exercise 2.2.7
Define, test, and debug the following procedures. Assume that *s* is any symbol, *n* is a nonnegative integer, *lst* is a list, *v* is a vector, *los* is a list of symbols, *vos* is a vector of symbols, *slst* is an s-list, and x is any object; and similarly *s1* is a symbol, *los2* is a list of symbols, *x1* is an object, etc. Make no other assumptions about the data unless further restrictions are given as part of a particular problem. You do not have to check that the input matches the description; for each procedure, assume that its input values are members of the specified data types.

To test your procedures, at the very minimum try all of the given examples. You should also use other examples to test your procedures, since the given examples are not adequate to reveal all possible errors.

1. (duple n x) returns a list containing *n* copies of x.

```
> (duple 2 3)
(3 3)
> (duple 4 '(ho ho))
((ho ho) (ho ho) (ho ho) (ho ho))
> (duple 0 '(blah))
()
```

2. (invert lst), where *lst* is a list of 2-lists (lists of length two), returns a list with each 2-list reversed.

```
> (invert '((a 1) (a 2) (b 1) (b 2)))
((1 a) (2 a) (1 b) (2 b))
```

3. (list-index s los) returns the zero-based index of the first occurrence of *s* in *los*, or -1 if there is no occurrence of *s* in *los*.

```
> (list-index 'c '(a b c d))
2
> (list-ref '(a b c) (list-index 'b '(a b c)))
b
```

4. (vector-index s vos) returns the zero-based index of the first occurrence of *s* in *vos*, or -1 if there is no occurrence of *s* in *vos*.

```
> (vector-index 'c '#(a b c d))
2
> (vector-ref '#(a b c) (vector-index 'b '#(a b c)))
b
```

5. (ribassoc s los v fail-value) returns the value in *v* that is associated with *s*, or *fail-value* if there is no associated value. If the first occurrence of *s* in *los* has index *n*, the value associated with *s* is the n^{th}nth value in *v*. There is no associated value for *s* if *s* is not a member of *los*. You may assume that *los* and *v* are the same length.

```
> (ribassoc 'b '(a b c) '#(1 2 3) 'fail)
2
> (ribassoc 'c '(a b foo) '#(3 squiggle bar) 'fail)
fail
> (ribassoc 'i '(a i o i) '#(fx (fz) () (fm fe)) 'fail)
(fz)
```

6. (filter-in p lst), where *p* is a predicate, returns the list of those elements in *lst* that satisfy the predicate.

```
> (filter-in number? '(a 2 (1 3) b 7))
(2 7)
> (filter-in symbol? '(1 2 (b c) 17 foo))
(a foo)
```

7. (product los1 los2) returns a list of 2-lists that represents the Cartesian product of los1 and los2. The 2-lists may appear in any order.

```
> (product '(a b c) '(x y))
((a x) (b y) (b x) (b y) (c x) (c y))
```

8. (swapper s1 s2 slst) returns a list the same as *slst*, but with all occurrences of *s1* replaced by *s2* and all occurrences of *s2* replaced by *s1*.

```
> (swapper 'a 'd '(a b c d))
(d b c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

9. (rotate los) returns a list similar to *los*, except that the last element of *los* becomes the first in the returned list.

```
> (rotate '(a b c d))
(d a b c)
> (rotate '(notmuch))
(notmuch)
> (rotate '())
```

```
()
```

•• Exercise 2.2.8

These are a bit harder.

1. (down lst) wraps parentheses around each top-level element of *lst*.

```
> (down '(1 2 3))
((1) (2) (3))
> (down '(a (more (complicated)) object))
((a) ((more (complicated))) (object))
```

2. (up lst) removes a pair of parentheses from each top-level element of *lst*. If a top-level element is not a list, it is included in the result, as is. The value of (up (down lst)) is equivalent to *lst*, but (down (up lst)) is not necessarily lst.

```
> (up '((1 2) (3 4)))
(1 2 3 4)
> (up '((x (y)) z))
(x (y) z)
```

3. (count-occurrences s slst) returns the number of occurrences of *s* in *slst*.

```
> (count-occurrences 'x '((f x) y (((x z) x))))
3
> (count-occurrences 'w '((f x) y (((x z) x))))
0
```

4. (flatten slst) returns a list of the symbols contained in *slst* in the order in which they occur when *slst* is printed. Intuitively, *flatten* removes all the inner parentheses from its argument.

```
> (flatten '(a b c))
(a b c)
> (flatten '((a b) c (((d)) e)))
(a b c d e)
> (flatten '(a b (() (c))))
(a b c)
```

5. (merge lon1 lon2), where *lon1* and *lon2* are lists of numbers that are sorted in ascending order, returns a sorted list of all the numbers in *lon1* and *lon2*.

```
> (merge '(1 4) '(1 2 8))
(1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 89==90))
(3 35 62 81 83 85 90 90 91)
```

□□

•• Exercise 2.2.9
These are harder still:

1. (path n bst), where *n* is a number and *bst* is a binary search tree that contains the number *n*, returns a list of *Ls* and *Rs* showing how to find the node containing *n*. If *n* is found at the root, it returns the empty list.

```
> (path 17 '(14 (7 () (12 () ()))
                (26 (20 (17 () ())
                        ())
                    (31 () ()))))
(R L L)
```

2. (car&cdr s slst errvalue) returns an expression that, when evaluated, produces the code for a procedure that takes a list with the same structure as *slst* and returns the value in the same position as the leftmost occurrence of *s* in *slst*. If *s* does not occur in *slst*, then *errvalue* is returned.

```
> (car&cdr 'a '(a b c) 'fail)
(lambda (lst) (car lst))
> (car&cdr 'c '(a b c) 'fail)
(lambda (lst) (car (cdr (cdr lst))))
> (car&cdr 'dog '(cat lion (fish dog) pig) 'fail)
(lambda (lst) (car (cdr (car (cdr (cdr lst))))))
> (car&cdr 'a '(b c) 'fail)
fail
```

3. (car&cdr2 s slst errvalue) is like the previous exercise, but it generates procedure compositions.

```
> (car&cdr2 'a '(a b c) 'fail)
car
> (car&cdr2 'c '(a b c) 'fail)
(compose car (compose cdr cdr))
> (car&cdr2 'dog '(cat lion (fish dog) pig) 'fail)
(compose car (compose cdr (compose car (compose cdr cdr))))
> (car&cdr2 'a '(b c) 'fail)
fail
```

4. (compose p1 ... pn), where p1,...pn is a sequence of zero or more procedures of one argument, returns the composition of all the procedures. The composition of zero procedures is the identity procedure, the composition of one procedure is the procedure itself, and the composition of two or more procedures is specified by this equation:

$$((compose\ p1\ p2\ \ldots)\ x) \Rightarrow (p1\ ((compose(p2\ \ldots)\ x))$$

((compose p1 p2 ...) x) ⇒ (p1 ((compose(p2 ...) x))

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

5. (sort lon) returns a list of the elements of *lon* in increasing order.

```
> (sort '(8 2 5 2 3))
(2 2 3 5 8)
```

6. (sort predicate lon) returns a list of elements determined by the predicate.

```
> (sort < '(8 2 5 2 3))
(2 2 3 5 8)
> (sort > '(8 2 5 2 3))
(8 5 3 2 2)
```

2.3 Static Properties of Variables

Those properties of a program that can be determined by analyzing the text of a program are said to be *static*, as opposed to the *dynamic* properties that are determined by run-time inputs. It is important to determine if a property is static, because static properties can be analyzed by a compiler to detect errors before run time and to improve the efficientcy of object code.

In Scheme, as in most other languages, the relation between a variable reference and the formal parameter to which it refers is a static property. In this section we focus on this relation and some of its important consequences.

2.3.1 Free and Bound Variables

In order to foucs on varialbe binding with a minimum of distraction, we initially study it in the most abstract context possible. For this purpose we introduce a language that has only variable references, *lambda* expressions with a single formal parameter, and procedure calls.

$$\begin{aligned}\langle exp\rangle ::= &\langle varref\rangle\\ &|(lambda(\langle var\rangle)\langle exp\rangle)\\ &|(\langle exp\rangle\ \langle exp\rangle)\end{aligned}$$

⟨exp⟩ ::=⟨varref⟩|(lambda(⟨var⟩)⟨exp⟩)|(⟨exp⟩ ⟨exp⟩)

This language is called the *lambda calculus. Although quite concise, its concepts generalize easily to most programming languages. For these reasons, the lambda calculus is the formal basis for much of the theory of programming languages.

The traditional syntax for procedures in the lambda calculus uses the Greek letter λλ (lambda), replacing the second alternative in the above grammar with

$$\lambda\langle var\rangle.\langle exp\rangle$$

λ⟨var⟩.⟨exp⟩

We use the keyword **lambda** and the extra parentheses so that these expressions look like Scheme expressions. Furthermore, since elements of $\langle exp\rangle$⟨exp⟩ are lists, it is convenient to write porgrams that manipulate them.

A variable reference is said to be *bound* in an expression if it refers to a formal parameter introduced in the expression. A reference that is not bound to a formal parameter in the expression is said to be *free*. Thus in

$$((lambda(x)\ x)\ y) \qquad (*)$$

((lambda(x) x) y) (*)

the reference to x is bound and the reference to y is free. A variable is said to *occur bound* in an expression if the expression contains a bound reference to the varialbe. Similaryly, a variable is said to *occur free* in an expression if the expression contains a free reference to the variable.

All variable references must have some associated value when they are evaluated at run time. If they are bound to a formal parameter, they are said to be *lexically bound*. Otherwise , they must either be bound at top level by definitions or be supplied by the system. In this case, they are said to be *globally bound*. It is an error to refernce a variable that is neither lexically nor globally bound.

The value of an expression depends only on the value associated with the variables that occur free within the expression. The context that surrounds the expression must provide these values. For example, the value of $(*)$(*) depends on the value of the free variable y. If $(*)$(*) were embeded in the body of a lambda expression with formal parameter y, as in

$$(lambda(y)\ ((lambda(x)\ x)\ y)) \qquad (**)$$

(lambda(y) ((lambda(x) x) y)) (**)

then the binding of this parameter would provide the value for the reference to variable y. Thus a variable reference that is free in one context, such as $(*)$(*) , may be bound in a larger surrounding context, such as $(**)$(**).

The value of an expression is independent of binding for variables that do not occur free in the expression. For example, the value of $(*)$(*) is independent of any bindings that might exist for x at the time that $(*)$(*) is evaluated. By the time the free occurrence of x in the body of $(lambda(x)\ x)$(lambda(x) x) is evaluated, it will have a new binding (in $(*)$(*), the value associated with y).

The meaning of $(lambda(x)\ x)$(lambda(x) x) is always the same: it is the identity function that returns whatever value it is passed. Other lambda expression without free variables also have fixed meanings. For example, the value of

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

is a procedure that takes a procedure *f*, and returns a procedure that takes a value *x*, applies *f* to it, and returns the result. Lambda expressions without free variables are called *combinators*. A few combinators, such as the identity function and the above application combinator, are useful programming tools. We shall use more elaborate combinators in the procedural representation of data types, beginning in section 3.6.

Free and bound occurrences may be defined formally as follows:

A variable x *occurs free* in an expression E if and only if

1. E is a variable reference and E is the same as x; or
2. E is of the form (E1 E2) and x occurs free in E_1E1 or E_2E2; or
3. E is of the form (lambda (y) E'E′), where *y* is different from x and x occurs free in E'E′.

A variable x *occurs bound* in an expression E if and only if

1. E is of the form (E1 E2) and x occurs bound in E_1E1 or E_2E2; or
2. E is of the form (lambda (y) E'E′), where x occurs bound in E'E′ or x and y are the same variable and y occurs free in E'E′.

No variable occurs bound in an expression consisting of just a single variable.

•• Exercise 2.3.1

Write a procedure *free-vars* that takes a list structure representing an expression in the lambda calculus syntax given above and returns a set, a list without duplicates, of all the variables that occur free in the expression. Similarly, write a procedure *bound-vars* that returns a set of all the variables that occur bound in its argument.

Hint: The definitions of occurs free and occurs bound are recursive and based on the structure of an expression. Your program should have a similar structure. □□

•• Exercise 2.3.3

Give an example of a lambda calculus expression in which the same variable occurs both bound and free. □□

•• Exercise 2.3.4

Give an example of a lambda calculus expression in which a variable occurs free but which has a value that is independent of the value of the free variable. □□

∘∘ Exercise 2.3.5

Scheme **lambda** expression may have any number of formal parameters, and Scheme procedure calls may have any number of operands. Modify the definitions of occurs free and occurs bound to allow **lambda** expressions with any number of formal parameters and procedure calls with any number of operands. □□

∘∘ Exercise 2.3.6
Extend the formal definitions of occurs free and occurs bound to include if expressions. □□

∘∘ Exercise 2.3.7
What effect does quote have on the set of free and bound variables? □□

2.3.2 Scope and Lexical Address