Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 1: Inductive Sets of Data

This chapter introduces the basic programming tools we will need to write interpreters, checkers and similar programs that form the heart of a programming language processor.

Because the syntax of a program in a language is usually a nested or tree-like structure, recursion will be at the core of our techniques. Section 1.1 and section 1.2 introduce methods for inductively specifying data structures and show how such specifications may be used to guide the construction of recursive programs. Section 1.3 shows how to extend these techniques to more complex problems. The chapter concludes with an extensive set of exercises. These 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.

1.1 Recursively Specified Data

When writing code for a procedure, we must know precisely what kinds of values may occur as arguments to the procedure, and what kinds of values are legal for the procedure to return. Often these sets of values are complex. In this section we introduce formal techniques for specifying sets of values.

1.1.1 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 S of the natural numbers $N = \{0, 1, 2, \ldots\}$.

Definition 1.1.1 *A natural number n is in S if and only if*

1. $n = 0$, *or*
2. $n - 3 \in S$.

Let us see how we can use this definition to determine what natural numbers are in S. We know that $0 \in S$. Therefore $3 \in S$, since $(3 - 3) = 0$ and $0 \in S$. Similarly $6 \in S$, since $(6–3) = 3$ and $3 \in S$. Continuing in this way, we can conclude that all multiples of 3 are in S.

What about other natural numbers? Is $1 \in S$? We know that $1 \neq 0$, so the first condition is not satisfied. Furthermore, $(1 - 3) = -2$, which is not a natural number and thus is not a member of S. Therefore the second condition is not satisfied. Since 1 satisfies neither condition, $1 \notin S$. Similarly, $2 \notin S$. What about 4? $4 \in S$ only if $1 \in S$. But $1 \notin S$, so $4 \notin S$, as well. Similarly, we can conclude that if n is a natural number and is not a multiple of 3, then $n \notin S$.

From this argument, we conclude that S is the set of natural numbers that are multiples of 3.

We can use this definition to write a procedure to decide whether a natural number n is in S.

```
in-S? : N → Bool
usage:  (in-S? n) = #t if n is in S, #f otherwise
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (if (>= (- n 3) 0)
        (in-S? (- n 3))
        #f))))
```

Here we have written a recursive procedure in Scheme that follows the definition. The notation **in-S?** : $N \rightarrow Bool$ is a comment, called the *contract* for this procedure. It means that in-S? is intended to be a procedure that takes a natural number and produces a boolean. Such comments are helpful for reading and writing code.

To determine whether $n \in S$, we first ask whether $n = 0$. If it is, then the answer is true. Otherwise we need to see whether $n - 3 \in S$. To do this, we first check to see whether $(n - 3) \geq 0$. If it is, we then can use our procedure to see whether it is in S. If it is not, then n cannot be in S.

Here is an alternative way of writing down the definition of S.

Definition 1.1.2 *Define the set S to be the smallest set contained in N and satisfying the following two properties:*

1. $0 \in S$, *and*
2. *if* $n \in S$, *then* $n + 3 \in S$.

A "smallest set" is the one that satisfies properties 1 and 2 and that is a subset of any other set satisfying properties 1 and 2. It is easy to see that there can be only one such set: if S_1 and S_2 both satisfy properties 1 and 2, and both are smallest, then $S_1 \subseteq S_2$ (since S_1 is smallest), and $S_2 \subseteq S_1$ (since S_2 is smallest), hence $S_1 = S_2$. We need this extra condition, because otherwise there are many sets that satisfy the remaining two conditions (see exercise 1.3).

Here is yet another way of writing the definition:

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

This is simply a shorthand notation for the preceding version of the definition. Each entry is called a *rule of inference*, or just a *rule*; the horizontal line is read as an "if-then." The part above the line is called the *hypothesis* or the *antecedent*; the part below the line is called the *conclusion* or the *consequent*. When there are two or more hypotheses listed, they are connected by an implicit "and" (see definition 1.1.5). A rule with no hypotheses is called an *axiom*. We often write an axiom without the horizontal line, like

$$0 \in S$$

The rules are interpreted as saying that a natural number n is in S if and only if the statement "$n \in S$" can be derived from the axioms by using the rules of inference finitely many times. This interpretation automatically makes S the smallest set that is closed under the rules.

These definitions all say the same thing. We call the first version a *top-down* definition, the second version a *bottom-up* definition, and the third version a *rules-of-inference* version.

Let us see how this works on some other examples.

Definition 1.1.3 (list of integers, top-down) *A Scheme list is a* list of integers *if and only if either*

1. *it is the empty list, or*
2. *it is a pair whose car is an integer and whose cdr is a list of integers.*

We use *Int* to denote the set of all integers, and *List-of-Int* to denote the set of lists of integers.

Definition 1.1.4 (list of integers, bottom-up) *The set List-of-Int is the smallest set of Scheme lists satisfying the following two properties:*

1. $() \in$ *List-of-Int, and*
2. *if* $n \in$ *Int and* $l \in$ *List-of-Int, then* $(n\ .\ l) \in$ *List-of-Int.*

Here we use the infix "." to denote the result of the cons operation in Scheme. The phrase $(n\ .\ l)$ denotes a Scheme pair whose car is n and whose cdr is l.

Definition 1.1.5 (list of integers, rules of inference)

$$() \in \mathit{List\text{-}of\text{-}Int}$$

$$\frac{n \in \mathit{Int} \qquad l \in \mathit{List\text{-}of\text{-}Int}}{(n\ .\ l) \in \mathit{List\text{-}of\text{-}Int}}$$

These three definitions are equivalent. We can show how to use them to generate some elements of *List-of-Int*.

1. () is a list of integers, because of property 1 of definition 1.1.4 or the first rule of definition 1.1.5.
2. (14 . ()) is a list of integers, because of property 2 of definition 1.1.4, since 14 is an integer and () is a list of integers. We can also write this as an instance of the second rule for *List-of-Int*.

$$\frac{14 \in Int \qquad () \in List\text{-}of\text{-}Int}{(14 \ . \ ()) \in List\text{-}of\text{-}Int}$$

3. (3 . (14 . ())) is a list of integers, because of property 2, since 3 is an integer and (14 . ()) is a list of integers. We can write this as another instance of the second rule for *List-of-Int*.

$$\frac{3 \in Int \qquad (14 \ . \ ()) \in List\text{-}of\text{-}Int}{(3 \ . \ (14 \ . \ ())) \in List\text{-}of\text{-}Int}$$

4. (-7 . (3 . (14 . ()))) is a list of integers, because of property 2, since -7 is a integer and (3 . (14 . ())) is a list of integers. Once more we can write this as an instance of the second rule for *List-of-Int*.

$$\frac{-7 \in Int \qquad (3 \ . \ (14 \ . \ ())) \in List\text{-}of\text{-}Int}{(-7 \ . \ (3 \ . \ (14 \ . \ ()))) \in List\text{-}of\text{-}Int}$$

5. Nothing is a list of integers unless it is built in this fashion.

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

We can also combine the rules to get a picture of the entire chain of reasoning that shows that (-7 . (3 . (14 . ()))) $\in$ *List-of-Int*. The tree-like picture below is called a *derivation* or *deduction tree*.

$$\dfrac{-7 \in N \qquad \dfrac{3 \in N \qquad \dfrac{14 \in N \qquad () \in List\text{-}of\text{-}Int}{(14 \ . \ ()) \in List\text{-}of\text{-}Int}}{(3 \ . \ (14 \ . \ ())) \in List\text{-}of\text{-}Int}}{(-7 \ . \ (3 \ . \ (14 \ . \ ()))) \in List\text{-}of\text{-}Int}$$

Exercise 1.1 [*] Write inductive definitions of the following sets. Write each definition in all three styles (top-down, bottom-up, and rules of inference). Using your rules, show the derivation of some sample elements of each set.

1. $\{3n + 2 \mid n \in N\}$
2. $\{2n + 3m + 1 \mid n, m \in N\}$
3. $\{(n, 2n + 1) \mid n \in N\}$
4. $\{(n, n^2) \mid n \in N\}$ Do not mention squaring in your rules. As a hint, remember the equation $(n + 1)^2 = n^2 + 2n + 1$.

Exercise 1.2 [**] What sets are defined by the following pairs of rules? Explain why.

1. $(0, 1) \in S$ $\dfrac{(n,k) \in S}{(n+1,k+7) \in S}$
2. $(0, 1) \in S$ $\dfrac{(n,k) \in S}{(n+1,2k) \in S}$
3. $(0, 0, 1) \in S$ $\dfrac{(n,i,j) \in S}{(n+1,j,i+j) \in S}$
4. [***] $(0, 1, 0) \in S$ $\dfrac{(n,i,j) \in S}{(n+1,i+2,i+j) \in S}$

Exercise 1.3 [*] Find a set T of natural numbers such that $0 \in T$, and whenever $n \in T$, then $n + 3 \in T$, but $T \neq S$, where S is the set defined in definition 1.1.2.

1.1.2 Defining Sets Using Grammars

The previous examples have been fairly straightforward, but it is easy to imagine how the process of describing more complex data types becomes quite cumbersome. To help with this, we show how to specify sets with *grammars*. Grammars are typically used to specify sets of strings, but we can use them to define sets of values as well.

For example, we can define the set *List-of-Int* by the grammar

```
List-of-Int ::= ()
List-of-Int ::= (Int . List-of-Int)
```

Here we have two rules corresponding to the two properties in definition 1.1.4 above. The first rule says that the empty list is in *List-of-Int*, and the second says that if *n* is in *Int* and *l* is in *List-of-Int*, then (*n* . *l*) is in *List-of-Int*. This set of rules is called a *grammar*.

Let us look at the pieces of this definition. In this definition we have

- **Nonterminal Symbols.** These are the names of the sets being defined. In this case there is only one such set, but in general, there might be several sets being defined. These sets are sometimes called *syntactic categories*.

  We will use the convention that nonterminals and sets have names that are capitalized, but we will use lower-case names when referring to their elements in prose. This is simpler than it sounds. For example, *Expression* is a nonterminal, but we will write *e* ∈ *Expression* or "*e* is an expression."

  Another common convention, called *Backus-Naur Form* or *BNF*, is to surround the word with angle brackets, e.g. ⟨expression⟩.

- **Terminal Symbols.** These are the characters in the external representation, in this case ., (, and). We typically write these using a typewriter font, e.g. lambda.

- **Productions.** The rules are called *productions*. Each production has a left-hand side, which is a nonterminal symbol, and a right-hand side, which consists of terminal and nonterminal symbols. The left- and right-hand sides are usually separated by the symbol ::=, read *is* or *can be*. The right-hand side specifies a method for constructing members of the syntactic category in terms of other syntactic categories and *terminal symbols,* such as the left parenthesis, right parenthesis, and the period.

Often some syntactic categories mentioned in a production are left undefined when their meaning is sufficiently clear from context, such as *Int*.

Grammars are often written using some notational shortcuts. It is common to omit the left-hand side of a production when it is the same as the left-hand side of the preceding production. Using this convention our example would be written as

```
List-of-Int ::= ()
            ::= (Int . List-of-Int)
```

One can also 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 *or*). The grammar for *List-of-Int* could be written

using " | "as

```
List-of-Int ::= () | (Int . List-of-Int)
```

Another shortcut is the *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. Using the Kleene star, the definition of *List-of-Int* is simply

```
List-of-Int ::= ({Int}*)
```

This includes the possibility of no instances at all. If there are zero instances, we get the empty string.

A variant of the star notation is *Kleene plus* {…}+, which indicates a sequence of *one* or more instances. Substituting + for * in the example above would define the syntactic category of non-empty lists of integers.

Still another variant of the star notation is the *separated list* notation. For example, we write $\{Int\}^{*(c)}$ to denote a sequence of any number of instances of the nonterminal *Int*, separated by the non-empty character sequence *c*. This includes the possibility of no instances at all. If there are zero instances, we get the empty string. For example, $\{Int\}^{*(,)}$ includes the strings

```
8
14, 12
7, 3, 14, 16
```

and $\{Int\}^{*(;)}$ includes the strings

```
8
14; 12
7; 3; 14; 16
```

These notational shortcuts are not essential. It is always possible to rewrite the grammar without them.

If a set is specified by a grammar, a *syntactic derivation* may be used to show that a given data value is a member of the set. Such a derivation starts with the nonterminal corresponding to the set. At each step, indicated by an arrow ⇒, a nonterminal is replaced 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 . ()) is a list of integers may be formalized with the syntactic derivation

```
    List-of-Int
⇒ (Int  . List-of-Int)
⇒ (14 . List-of-Int)
⇒ (14 . ())
```

The order in which nonterminals are replaced does not matter. Thus, here is another derivation of (14 . ()).

```
    List-of-Int
⇒ (Int . List-of-Int)
⇒ (Int .())
⇒ (14 . ())
```

Exercise 1.4 [*] Write a derivation from *List-of-Int* to (-7 . (3 . (14 . ()))).

Let us consider the definitions of some other useful sets.

1. Many symbol manipulation procedures are designed to operate on lists that contain only symbols and other similarly restricted lists. We call these lists *s-lists*, defined as follows:

   Definition 1.1.6 (s-list, s-exp)

   ```
   S-list ::= ({S-exp}*)
   S-exp ::= Symbol | S-list
   ```

   An s-list is a list of s-exps, and an s-exp is either an s-list or a symbol. Here are some s-lists.

   ```
   (a b c)
   (an (((s-list)) (with () lots) ((of) nesting)))
   ```

   We may occasionally use an expanded definition of s-list with integers allowed, as well as symbols.

2. A binary tree with numeric leaves and interior nodes labeled with symbols may be represented using three-element lists for the interior nodes by the grammar:

   Definition 1.1.7 (binary tree)

   ```
   Bintree ::= Int | (Symbol Bintree Bintree)
   ```

   Here are some examples of such trees:

   ```
   1
   2
   (foo 1 2)
   (bar 1 (foo 1 2))
   (baz
     (bar 1 (foo 1 2))
     (biz 4 5))
   ```

3. The *lambda calculus* is a simple language that is often used to study the theory of programming languages. This language consists only of variable references, procedures that take a single argument, and procedure calls. We can define it with the grammar:

   Definition 1.1.8 (lambda expression)

   ```
   LcExp ::= Identifier
             ::= (lambda (Identifier) LcExp)
             ::= (LcExp LcExp)
   ```

   where an identifier is any symbol other than lambda.

   The identifier in the second production is the name of a variable in the body of the lambda expression. This variable is called the *bound variable* of the expression, because it binds or captures any occurrences of the variable in the body. Any occurrence of that variable in the body refers to this one.

   To see how this works, consider the lambda calculus extended with arithmetic operators. In that language,

   ```
   (lambda (x) (+ x 5))
   ```

   is an expression in which x is the bound variable. This expression describes a procedure that adds 5 to its argument. Therefore, in

   ```
   ((lambda (x) (+ x 5)) (- x 7))
   ```

   the last occurrence of x does not refer to the x that is bound in the lambda expression. We discuss this in section 1.2.4, where we introduce occurs-free?.

   This grammar defines the elements of *LcExp* as Scheme values, so it becomes easy to write programs that manipulate them.

These grammars 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. Consider binary search trees. A node in a binary search tree is either empty or contains an integer and two subtrees

```
Binary-search-tree ::= () | (Int Binary-search-tree Binary-search-tree)
```

This correctly describes the structure of each node but ignores an important fact about binary search trees: 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.

Because of this additional constraint, not every syntactic derivation from *Binary-search-tree* leads to a correct binary search tree. To determine whether a particular production can be applied in a particular syntactic derivation, we have to look at the context in which the production is applied. Such constraints are called *context-sensitive constraints* or *invariants*.

Context-sensitive constraints also arise when specifying the syntax of programming languages. For instance, in many languages every variable must be declared before it is used. This constraint on the use of variables is 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 the ones we consider in this chapter. In practice, the usual approach is first to specify a context-free grammar. Context-

sensitive constraints are then added using other methods. We show an example of such techniques in chapter 7.

1.1.3 Induction

Having described sets inductively, we can use the inductive definitions in two ways: to prove theorems about members of the set and to write programs that manipulate them. Here we present an example of such a proof; writing the programs is the subject of the next section.

Theorem 1.1.1 *Let t be a binary tree, as defined in definition 1.1.7. Then t contains an odd number of nodes.*

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

1. There are no trees with 0 nodes, so *IH*(0) holds trivially.
2. Let *k* be an integer such that *IH*(*k*) holds, that is, any tree with ≤ *k* nodes actually has an odd number of nodes. We need to show that *IH*(*k* + 1) holds as well: that any tree with ≤ *k* + 1 nodes has an odd number of nodes. If *t* has ≤ *k* + 1 nodes, there are exactly two possibilities according to the definition of a binary tree:
   a. *t* could be of the form *n*, where *n* is an integer. In this case, *t* has exactly one node, and one is odd.
   b. *t* could be of the form (*sym* t_1 t_2),where *sym* is a symbol and t_1 and t_2 are trees. Now t_1 and t_2 must have fewer nodes than *t*. Since *t* has ≤ *k* + 1 nodes, t_1 and t_2 must have ≤ *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. Hence the total number of nodes in the tree, counting the two subtrees and the root, is

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

      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 *t* are always smaller than *t* itself. This pattern of proof is called *structural induction*.

> **Proof by Structural Induction**
>
> To prove that a proposition IH(s) is true for all structures s, prove the following:
>
> 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.

Exercise 1.5 [**] Prove that if *e* ∈ *LcExp*, then there are the same number of left and right parentheses in *e*.

1.2 Deriving Recursive Programs

We have used the method of inductive definition to characterize complicated sets. We have seen that we can analyze an element of an inductively defined set to see how it is built from smaller elements of the set. We have used this idea to write a procedure in-S? to decide whether a natural number is in the set *S*. We now use the same idea to define more general procedures that compute on inductively defined sets.

Recursive procedures rely on an important principle:

> **The Smaller-Subproblem Principle**
>
> If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the problem to solve the subproblem.

The solution returned 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.

We illustrate this idea with a sequence of examples.

1.2.1 list-length

The standard Scheme procedure length determines the number of elements in a list.

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

Let us write our own procedure, called list-length, that does the same thing.

We begin by writing down the *contract* for the procedure. The contract specifies the sets of possible arguments and possible return values for the procedure. The contract also may include the intended usage or behavior of the procedure. This helps us keep track of our intentions both as we write and afterwards. In code, this would be a comment; we typeset it for readability.

```
list-length : List → Int
usage:   (list-length l) = the length of l
(define list-length
  (lambda (lst)
    ...))
```

We can define the set of lists by

```
List ::= () | (Scheme value . List)
```

Therefore we consider each possibility for a list. If the list is empty, then its length is 0.

```
list-length : List → Int
usage: (list-length l) = the length of l
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      ...)))
```

If a list is non-empty, then its length is one more than the length of its cdr. This gives us a complete definition.

```
list-length : List → Int
usage: (list-length l) = the length of l
(define list-length
  (lambda (lst)
    (if (null? lst)
      0
      (+ 1 (list-length (cdr lst))))))
```

We can watch list-length compute by using its definition.

```
(list-length '(a (b c) d))
= (+ 1 (list-length '((b c) d)))
= (+ 1 (+ 1 (list-length '(d))))
= (+ 1 (+ 1 (+ 1 (list-length '()))))
= (+ 1 (+ 1 (+ 1 0)))
=3
```

1.2.2 nth-element

The standard Scheme procedure list-ref takes a list lst and a zero-based index n and returns element number n of lst.

```
> (list-ref '(a b c) 1)
b
```

Let us write our own procedure, called nth-element, that does the same thing.

Again we use the definition of *List* above.

What should (nth-element *lst n*) return when *lst* is empty? In this case, (nth-element *lst n*) is asking for an element of an empty list, so we report an error.

What should (nth-element *lst n*) return when *lst* is non-empty? The answer depends on *n*. If *n* = 0, the answer is simply the car of *lst*.

What should (nth-element *lst n*) return when *lst* is non-empty and $n \neq 0$? In this case, the answer is the $(n - 1)$-st element of the cdr of *lst*. Since $n \in N$ and $n \neq 0$, we know that $n - 1$ must also be in N, so we can find the $(n - 1)$-st element by recursively calling nth-element.

This leads us to the definition

```
nth-element : List × Int → SchemeVal
usage: (nth-element lst n) = the n-th element of lst
(define nth-element
  (lambda (lst n)
    (if (null? lst)
      (report-list-too-short n)
      (if (zero? n)
        (car lst)
        (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element
      "List too short by ~s elements.~%" (+ n 1))))
```

Here the notation **nth-element** : *List* × *Int*→ *SchemeVal* means that **nth-element** is a procedure that takes two arguments, a list and an integer, and returns a Scheme value. This is the same notation that is used in mathematics when we write $f : A \times B \rightarrow C$.

The procedure report-list-too-short reports an error condition by calling eopl:error. The procedure eopl:error aborts the computation. Its first argument is a symbol that allows the error message to identify the procedure that called eopl:error. The second argument is a string that is then printed in the error message. There must then be an additional argument for each instance of the character sequence ~s in the string. The values of these arguments are printed in place of the corresponding ~s when the string is printed. A ~% is treated as a new line. After the error message is printed, the computation is aborted. This procedure eopl:error is not part of standard Scheme, but most implementations of Scheme provide such a facility. We use procedures named report- to report errors in a similar fashion throughout the book.

Watch how nth-element computes its answer:

```
  (nth-element '(a b c d e) 3)
= (nth-element   '(b c d e) 2)
= (nth-element     '(c d e) 1)
= (nth-element       '(d e) 0)
= d
```

Here nth-element recurs on shorter and shorter lists, and on smaller and smaller numbers.

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 we received an error message from car, we might have to look for uses of car throughout our program.

Exercise 1.6 [*] If we reversed the order of the tests in nth-element, what would go wrong?

Exercise 1.7 [**] The error message from nth-element is uninformative. Rewrite nth-element so that it produces a more informative error message, such as “(a b c) does not have 8 elements.”

1.2.3 remove-first

The procedure remove-first should take two arguments: a symbol, *s*, and a list of symbols, *los*. It should return 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 writing the definition of this procedure, we must complete the problem specification by defining the set *List-of-Symbol* of lists of symbols. Unlike the s-lists introduced in the last section, these lists of symbols do not contain sublists.

```
List-of-Symbol ::= () | (Symbol . List-of-Symbol)
```

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.

```
remove-first : Sym × Listof(Sym) → Listof(Sym)
usage: (remove-first slos) 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.
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      ...)))
```

Here we have written the contract with *Listof* (*Sym*) instead of *List-of-Symbol*. This notation will allow us to avoid many definitions like the ones above.

If *los* is non-empty, is there some case where we can determine the answer immediately? If the first element of *los* is *s*, say *los* = (s s_1 … s_{n-1}), the first occurrence of *s* is as the first element of *los*. So the result of removing it is just (s_1 … s_{n-1}).

```
remove-first : Sym × Listof(Sym) → Listof(Sym)
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      (if (eqv? (car los) s)
        (cdr los)
        ...))))
```

If the first element of *los* is not *s*, say *los* = (s_0 s_1 … s_{n-1}), then we know that s_0 is not the first occurrence of *s*. Therefore the first element of the answer must be s_0, which is the value of the expression (car los). Furthermore, the first occurrence of *s* in *los* must be its first occurrence in (s_1 … s_{n-1}). So the rest of the answer must be the result of removing the first occurrence of *s* from the cdr of *los*. Since the cdr of *los* is shorter than *los*, we may recursively call remove-first to remove *s* from the cdr of *los*. So the cdr of the answer can be obtained as the value of (remove-first s (cdr los)). Since we know how to find the car and cdr of the answer, we can find the whole answer by combining them with cons, using the expression (cons (car los) (remove-first s (cdr los))). With this, the complete definition of remove-first becomes

```
remove-first : Sym × Listof(Sym) → Listof(Sym)
(define remove-first
  (lambda (s los)
    (if (null? los)
      '()
      (if (eqv? (car los) s)
        (cdr los)
        (cons (car los) (remove-first s (cdr los)))))))
```

Exercise 1.8 [*] In the definition of remove-first, if the last line were replaced by (remove-first s (cdr los)), what function would the resulting procedure compute? Give the contract, including the usage statement, for the revised procedure.

Exercise 1.9 [**] Define remove, which is like remove-first, except that it removes *all* occurrences of a given symbol from a list of symbols, not just the first.

1.2.4 occurs-free?

The procedure occurs-free? should take a variable *var*, represented as a Scheme symbol, and a lambda-calculus expression *exp* as defined in definition 1.1.8, and determine whether or not *var* occurs free in *exp*. We say that a variable *occurs free* in an expression *exp* if it has some occurrence in *exp* that is not inside some lambda binding of the same variable. For example,

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

We can solve this problem by following the grammar for lambda-calculus expressions

```
LcExp ::= Identifier
          ::= (lambda (Identifier) LcExp)
          ::= (LcExp LcExp)
```

We can summarize these cases in the rules:

- If the expression *e* is a variable, then the variable *x* occurs free in *e* if and only if *x* is the same as *e*.
- If the expression *e* is of the form (lambda (*y*) e'), then the variable *x* occurs free in *e* if and only if *y* is different from *x* and *x* occurs free in e'.
- If the expression *e* is of the form (e_1 e_2), then *x* occurs free in *e* if and only if it occurs free in e_1 or e_2. Here, we use “or” to mean *inclusive or*, meaning that this includes the possibility that *x* occurs free in both e_1 and e_2. We will generally use “or” in this sense.

You should convince yourself that these rules capture the notion of occurring “not inside a lambda-binding of *x*.”

Exercise 1.10 [*] We typically use “or” to mean “inclusive or.” What other meanings can “or” have?

Then it is easy to define occurs-free?. Since there are three alternatives to be checked, we use a Scheme cond rather than an if. In Scheme, (or exp_1 exp_2) returns a true value if either exp_1 or exp_2 returns a true value.

```
occurs-free? : Sym × LcExp → Bool
usage:   returns #t if the symbol var occurs free
         in exp, otherwise returns #f.
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
       (and
         (not (eqv? var (car (cadr exp))))
         (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```

This procedure is not as readable as it might be. It is hard to tell, for example, that (car (cadr exp)) refers to the declaration of a variable in a lambda expression, or that (caddr exp) refers to its body. We show how to improve this situation considerably in section 2.5.

1.2.5 subst

The procedure subst should take three arguments: two symbols, new and old, and an s-list, slist. All elements of slist are examined, and a new list is returned that is similar to slist 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 should reflect the definition of s-lists (definition 1.1.6)

```
S-list  ::= ({S-exp}*)
```

```
S-exp ::= Symbol | S-list
```

The Kleene star gives a concise description of the set of s-lists, but it is not so helpful for writing programs. Therefore our first step is to rewrite the grammar to eliminate the use of the Kleene star. The resulting grammar suggests that our procedure should recur on the car and cdr of an s-list.

```
S-list  ::= ()
          ::= (S-exp . S-list)
S-exp  ::= Symbol | S-list
```

This example is more complex than our previous ones because the grammar for its input contains two nonterminals, *S-list* and *S-exp*. Therefore we will have two procedures, one for dealing with *S-list* and one for dealing with *S-exp*:

```
subst : Sym × Sym × S-list → S-list
(define subst
  (lambda (new old slist)
    ...))

subst-in-s-exp : Sym × Sym × S-exp → S-exp
(define subst-in-s-exp
  (lambda (new old sexp)
    ...))
```

Let us first work on subst. If the list is empty, there are no occurrences of old to replace.

```
subst : Sym × Sym × S-list → S-list
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      ...)))
```

If slist is non-empty, its car is a member of *S-exp* and its cdr is another s-list. In this case, the answer should be a list whose car is the result of changing old to new in the car of slist, and whose cdr is the result of changing old to new in the cdr of slist. Since the car of slist is an element of *S-exp*, we solve the subproblem for the car using subst-in-s-exp. Since the cdr of slist is an element of *S-list*, we recur on the cdr using subst:

```
subst : Sym × Sym × S-list → S-list
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

Now we can move on to subst-in-s-exp. From the grammar, we know that the symbol expression sexp is either a symbol or an s-list. If it is a symbol, we need to ask whether it is the same as the symbol old. If it is, the answer is new; if it is some other symbol, the answer is the same as sexp. If sexp is an s-list, then we can recur using subst to find the answer.

```
subst-in-s-exp : Sym × Sym × S-exp → S-exp
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

Since we have strictly followed the definition of *S-list* and *S-exp*, this recursion is guaranteed to halt. Since subst and subst-in-s-exp call each other recursively, we say they are *mutually recursive*.

The decomposition 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 greatly simplifies our thinking about more complicated programs.

Exercise 1.11 [*] In the last line of subst-in-s-exp, the recursion is on sexp and not a smaller substructure. Why is the recursion guaranteed to halt?

Exercise 1.12 [*] Eliminate the one call to subst-in-s-exp in subst by replacing it by its definition and simplifying the resulting procedure. The result will be a version of subst that does not need subst-in-s-exp. This technique is called *inlining*, and is used by optimizing compilers.

Exercise 1.13 [**] In our example, we began by eliminating the Kleene star in the grammar for *S-list*. Write subst following the original grammar by using map.

We've now developed a recipe for writing procedures that operate on inductively defined data sets. We summarize it as a slogan.

> **Follow the Grammar!**
>
> When defining a procedure that operates on inductively defined data, the structure of the program should be patterned after the structure of the data.

More precisely:

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal, and nothing else.
- In each procedure, write one alternative for each production corresponding to that nonterminal. You may need additional case structure, but this will get you started. For each nonterminal that appears in the right-hand side, write a recursive call to the procedure for that nonterminal.

1.3 Auxiliary Procedures and Context Arguments

The *Follow-the-Grammar* recipe is powerful, but sometimes it is not sufficient. Consider the procedure number-elements. This procedure should take any list (v_0 v_1 v_2 ...) and return the list ((0 v_0) (1 v_1) (2 v_2) …).

A straightforward decomposition of the kind we've used so far does not solve this problem, because there is no obvious way to build the value of (number-elements lst) from the value of (number-elements (cdr lst)) (but see exercise 1.36).

To solve this problem, we need to *generalize* the problem. We write a new procedure number-elements-from that takes an additional argument *n* that specifies the number to start from. This procedure is easy to write, by recursion on the list.

```
number-elements-from : Listof(SchemeVal) × Int → Listof(List(Int, SchemeVal))

usage: (number-elements-from '(v0 v1 v2 ...) n)
        =((nv0)(n + 1 v1)(n + 2 v2) ...)
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-from (cdr lst) (+ n 1))))))
```

Here the contract header tells us that this procedure takes two arguments, a list (containing any Scheme values) and an integer, and returns a list of things, each of which is a list consisting of two elements: an integer and a Scheme value.

Once we have defined number-elements-from, it's easy to write the desired procedure.

```
number-elements : List → Listof(List(Int, SchemeVal))
(define number-elements
  (lambda (lst)
    (number-elements-from lst 0)))
```

There are two important observations to be made here. First, the procedure number-elements-from has a specification that is *independent* of the specification of number-elements. It's very common for a programmer to write a procedure that simply calls some auxiliary procedure with some additional constant arguments. Unless we can understand what that auxiliary procedure does for *every* value of its arguments, then we can't possibly understand what the calling procedure does. This gives us a slogan:

> **No Mysterious Auxiliaries!**
>
> When defining an auxiliary procedure, always specify what it does on all arguments, not just the initial values.

Second, the two arguments to number-elements-from play two different roles. The first argument is the list we are working on. It gets smaller at every recursive call. The second argument, however, is an abstraction of the *context* in which we are working. In this example, when we call number-elements, we end up calling number-elements-from on each sublist of the original list. The second argument tells us the position of the sublist in the original list. This need not decrease at a recursive call; indeed it

grows, because we are passing over another element of the original list. We sometimes call this a *context argument* or *inherited attribute*.

As another example, consider the problem of summing all the values in a vector.

If we were summing the values in a list, we could follow the grammar to recur on the cdr of the list. This would get us a procedure like

```
list-sum : Listof(Int) → Int
(define list-sum
  (lambda (loi)
    (if (null? loi)
      0
      (+ (car loi)
         (list-sum (cdr loi))))))
```

But it is not possible to proceed in this way with vectors, because they do not decompose as readily.

Since we cannot decompose vectors, we generalize the problem to compute the sum of part of the vector. The specification of our problem is to compute

$$\sum_{i=0}^{i=length(v)-1} v_i$$

where *v* is a vector of integers. We generalize it by turning the upper bound into a parameter *n*, so that the new task is to compute

$$\sum_{i=0}^{i=n} v_i$$

where $0 \le n < length(v)$.

This procedure is straightforward to write from its specification, using induction on its second argument *n*.

```
partial-vector-sum : Vectorof(Int) × Int → Int
usage: if 0 ≤ n < length(v), then
                                          i=n
                                          Σ   v_i
            (partial-vector-sum v n) =    i=0
(define partial-vector-sum
  (lambda (v n)
    (if (zero? n)
      (vector-ref v 0)
      (+ (vector-ref v n)
         (partial-vector-sum v (- n 1))))))
```

Since *n* decreases steadily to zero, a proof of correctness for this program would proceed by induction on *n*. Because $0 \le n$ and $n \neq 0$, we can deduce that $0 \le (n - 1)$, so that the recursive call to the procedure partial-vector-sum satisfies its contract.

It is now a simple matter to solve our original problem. The procedure partial-vector-sum doesn't apply if the vector is of length 0, so we need to handle that case separately.

```
vector-sum : Vectorof(Int) → Int
                                 i=length(v)-1
                                    Σ   v_i
usage: (vector-sum v) =             i=0
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v)))
      (if (zero? n)
        0
        (partial-vector-sum v (- n 1))))))
```

There are many other situations in which it may be helpful or necessary to introduce auxiliary variables or procedures to solve a problem. Always feel free to do so, provided that you can give an independent specification of what the new procedure is intended to do.

Exercise 1.14 [**] Given the assumption $0 \le n < length(v)$, prove that partial-vector-sum is correct.

1.4 Exercises

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

In each of these exercises, assume that s is a symbol, n is a nonnegative integer, lst is a list, loi is a list of integers, los is a list of symbols, slist is an s-list, and x is any Scheme value; and similarly s1 is a symbol, los2 is a list of symbols, x1 is a Scheme value, etc. Also assume that pred is a predicate, that is, a procedure that takes any Scheme value and always returns either #t or #f. Make no other assumptions about the data unless further restrictions are given as part of a particular problem. For these exercises, there is no need to check that the input matches the description; for each procedure, assume that its input values are members of the specified sets.

Define, test, and debug each procedure. Your definition should include a contract and usage comment in the style we have used in this chapter. Feel free to define auxiliary procedures, but each auxiliary procedure you define should have its own specification, as in section 1.3.

To test these procedures, first try all the given examples. Then use other examples to test these procedures, since the given examples are not adequate to reveal all possible errors.

Exercise 1.15 [*] (duple n x) returns a list containing n copies of x.

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

Exercise 1.16 [*] (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) (1 b) (2 b)))
((1 a) (2 a) (b 1) (b 2))
```

Exercise 1.17 [*] (down lst) wraps parentheses around each top-level element of lst.

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

Exercise 1.18 [*] (swapper s1 s2 slist) returns a list the same as slist, 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 'a 'd '(a d () c d))
(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

Exercise 1.19 [**] (list-set lst n x) returns a list like lst, except that the n-th element, using zero-based indexing, is x.

```
> (list-set '(a b c d) 2 '(1 2))
(a b (1 2) d)
> (list-ref (list-set '(a b c d) 3 '(1 5 10)) 3)
(1 5 10)
```

Exercise 1.20 [*] (count-occurrences s slist) returns the number of occurrences of s in slist.

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

Exercise 1.21 [**] (product sos1 sos2), where sos1 and sos2 are each a list of symbols without repetitions, returns a list of 2-lists that represents the Cartesian product of sos1 and sos2. The 2-lists may appear in any order.

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

Exercise 1.22 [**] (filter-in pred lst) returns the list of those elements in lst that satisfy the predicate pred.

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

Exercise 1.23 [**] (list-index pred lst) returns the 0-based position of the first element of lst that satisfies the predicate pred. If no element of lst satisfies the predicate, then list-index returns #f.

```
> (list-index number? '(a 2 (1 3) b 7))
1
> (list-index symbol? '(a (b c) 17 foo))
0
> (list-index symbol? '(1 2 (a b) 3))
#f
```

Exercise 1.24 [**] (every? pred lst) returns #f if any element of lst fails to satisfy pred, and returns #t otherwise.

```
> (every? number? '(a b c 3 e))
#f
> (every? number? '(1 2 3 5 4))
#t
```

Exercise 1.25 [**] (exists? pred lst) returns #t if any element of lst satisfies pred, and returns #f otherwise.

```
> (exists? number? '(a b c 3 e))
#t
> (exists? number? '(a b c d e))
#f
```

Exercise 1.26 [**] (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. (See exercise 1.17.)

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

Exercise 1.27 [**] (flatten slist) returns a list of the symbols contained in slist in the order in which they occur when slist is printed. Intuitively, flatten removes all the inner parentheses from its argument.

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

Exercise 1.28 [**] (merge loi1 loi2), where loi1 and loi2 are lists of integers that are sorted in ascending order, returns a sorted list of all the integers in loi1 and loi2.

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

Exercise 1.29 [**] (sort loi) returns a list of the elements of loi in ascending order.

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

Exercise 1.30 [**] (sort/predicate pred loi) returns a list of elements sorted by the predicate.

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

Exercise 1.31 [*] Write the following procedures for calculating on a bintree (definition 1.1.7): leaf and interior-node, which build bintrees, leaf?, which tests whether a bintree is a leaf, and lson, rson, and contents-of, which extract the components of a node. contents-of should work on both leaves and interior nodes.

Exercise 1.32 [*] Write a procedure double-tree that takes a bintree, as represented in definition 1.1.7, and produces another bintree like the original, but with all the integers in the leaves doubled.

Exercise 1.33 [**] Write a procedure mark-leaves-with-red-depth that takes a bintree (definition 1.1.7), and produces a bintree of the same shape as the original, except that in the new tree, each leaf contains the integer of nodes between it and the root that contain the symbol red. For example, the expression

```
(mark-leaves-with-red-depth
  (interior-node 'red
    (interior-node 'bar
      (leaf 26)
      (leaf 12))
    (interior-node 'red
      (leaf 11)
      (interior-node 'quux
        (leaf 117)
        (leaf 14))
```

which is written using the procedures defined in exercise 1.31, should return the bintree

```
(red
  (bar 1 1)
  (red 2 (quux 2 2)))
```

Exercise 1.34 [***] Write a procedure path that takes an integer n and a binary search tree bst (page 10) that contains the integer n, and returns a list of lefts and rights 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 () ()))))
(right left left)
```

Exercise 1.35 [***] Write a procedure number-leaves that takes a bintree, and produces a bintree like the original, except the contents of the leaves are numbered starting from 0. For example,

```
(number-leaves
  (interior-node 'foo
    (interior-node 'bar
      (leaf 26)
      (leaf 12))
    (interior-node 'baz
      (leaf 11)
      (interior-node 'quux
       (leaf 117)
       (leaf 14))
```

should return

```
(foo
  (bar 0 1)
  (baz
    2
    (quux 3 4)))
```

Exercise 1.36 [***] Write a procedure g such that number-elements from page 23 could be defined as

```
(define number-elements
  (lambda (lst)
    (if (null? lst) '()
      (g (list 0 (car lst)) (number-elements (cdr lst))))))
```