Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 2: Data Abstraction

2.1 Specifying Data via Interfaces

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.

The representation of these entities is often complex, so we do not want to be concerned with their details when we can avoid them. We may also decide to change the representation of the data. The most efficient representation is often a lot more difficult to implement, so we may wish to develop a simple implementation first and only change to a more efficient representation if it proves critical to the overall performance of a system. If we decide to change the representation of some data for any reason, we must be able to locate all parts of a program that are dependent on the representation. This is accomplished using the technique of *data abstraction*.

Data abstraction divides a data type into two pieces: an *interface* and an *implementation*. The interface tells us what the data of the type represents, what the operations on the data are, and what properties these operations may be relied on to have. The *implementation* provides a specific representation of the data and code for the operations that make use of that data representation.

A data type that is abstract in this way is said to be an *abstract data type*. The rest of the program, the *client* of the data type, manipulates the new data only through the operations specified in the interface. Thus if we wish to change the representation of the data, all we must do is change the implementation of the operations in the interface.

This is a familiar idea: when we write programs that manipulate files, most of the time we care only that we can invoke procedures that perform the open, close, read, and other typical operations on files. Similarly, most of the time, we don't care how integers are actually represented inside the machine. Our only concern is that we can perform the arithmetic operations reliably.

When the client manipulates the values of the data type only through the procedures in the interface, we say that the client code is *representation-independent*, because then the code does not rely on the representation of the values in the data type.

All the knowledge about how the data is represented must therefore reside in the code of the implementation. The most important part of an implementation is the specification of how the data is represented. We use the notation $\lceil v \rceil$ for "the representation of data *v*."

To make this clearer, let us consider a simple example: the data type of natural numbers. The data to be represented are the natural numbers. The interface is to consist of four procedures: zero, is-zero?, successor, and predecessor. Of course, not just any set of procedures will be acceptable as an implementation of this interface. A set of procedures will be acceptable as implementations of zero, is-zero?, successor, and predecessor only if they satisfy the four equations

$$\texttt{(zero)} = \lceil 0 \rceil$$

$$\texttt{(is-zero?} \lceil n \rceil\texttt{)} = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$\texttt{(successor} \lceil n \rceil\texttt{)} = \lceil n + 1 \rceil \quad (n \geq 0)$$

$$\texttt{(predecessor} \lceil n + 1 \rceil\texttt{)} = \lceil n \rceil \quad (n \geq 0)$$

This specification does not dictate how these natural numbers are to be represented. It requires only that these procedures conspire to produce the specified behavior. Thus, the procedure zero must return the representation of 0. The procedure successor, given the representation of the number *n*, must return the representation of the number *n* + 1, and so on. The specification says nothing about (predecessor (zero)), so under this specification any behavior would be acceptable.

We can now write client programs that manipulate natural numbers, and we are guaranteed that they will get correct answers, no matter what representation is in use. For example,

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
      y
      (successor (plus (predecessor x) y)))))
```

will satisfy (plus $\lceil x \rceil \lceil y \rceil$) = $\lceil x + y \rceil$, no matter what implementation of the natural numbers we use.

Most interfaces will contain some *constructors* that build elements of the data type, and some *observers* that extract information from values of the data type. Here we have three constructors, zero, successor, and predecessor, and one observer, is-zero?.

There are many possible representations of this interface. Let us consider three of them.

1. *Unary representation:* In the unary representation, the natural number *n* is represented by a list of *n* #t's. Thus, 0 is represented by (), 1 is represented by (#t), 2 is represented by (#t #t), etc. We can define this representation inductively by:

   - ⌈0⌉ = ()
   - ⌈*n* + 1⌉ = (#t . ⌈*n*⌉)

   In this representation, we can satisfy the specification by writing

   ```
   (define zero (lambda () '()))
   (define is-zero? (lambda (n) (null? n)))
   (define successor (lambda (n) (cons #t n)))
   (define predecessor (lambda (n) (cdr n)))
   ```

2. *Scheme number representation:* In this representation, we simply use Scheme's internal representation of numbers (which might itself be quite complicated!). We let ⌈*n*⌉ be the Scheme integer n, and define the four required entities by

   ```
   (define zero (lambda () 0))
   (define is-zero? (lambda (n) (zero? n)))
   (define successor (lambda (n) (+ n 1)))
   (define predecessor (lambda (n) (- n 1)))
   ```

3. *Bignum representation:* In the bignum representation, numbers are represented in base *N*, for some large integer *N*. The representation becomes a list consisting of numbers between 0 and *N* – 1 (sometimes called *bigits* rather than digits). This representation makes it easy to represent integers that are much larger than can be represented in a machine word. For our purposes, it is convenient to keep the list with least-significant bigit first. We can define the representation inductively by

   $$\lceil n \rceil = \begin{cases} () & n = 0 \\ (r \ . \ \lceil q \rceil) & n = qN + r,\ 0 \le r < N \end{cases}$$

   So if *N* = 16, then ⌈33⌉ = (1 2) and ⌈258⌉ = (2 0 1), since

   $$258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$$

   None of these implementations enforces data abstraction. There is nothing to prevent a client program from looking at the representation and determining whether it is a list or a Scheme integer. On the other hand, some languages provide direct support for data abstractions: they allow the programmer to create new interfaces and check that the new data is manipulated only through the procedures in the interface. If the representation of a type is hidden, so it cannot be exposed by any operation (including printing), the type is said to be *opaque.* Otherwise, it is said to be *transparent.*

   Scheme does not provide a standard mechanism for creating new opaque types. Thus we settle for an intermediate level of abstraction: we define interfaces and rely on the writer of the client program to be discreet and use only the procedures in the interfaces.

   In chapter 8, we discuss ways in which a language can enforce such protocols.

   Exercise 2.1 [*] Implement the four required operations for bigits. Then use your implementation to calculate the factorial of 10. How does the execution time vary as this argument changes? How does the execution time vary as the base changes? Explain why.

   Exercise 2.2 [**] Analyze each of these proposed representations critically. To what extent do they succeed or fail in satisfying the specification of the data type?

   Exercise 2.3 [**] Define a representation of all the integers (negative and nonnegative) as diff-trees, where a diff-tree is a

list defined by the grammar

```
Diff-tree ::= (one) | (diff Diff-tree Diff-tree)
```

The list (one) represents 1. If t_1 represents n_1 and t_2 represents n_2, then (diff t_1 t_2) is a representation of $n_1 - n_2$.

So both (one) and (diff (one) (diff (one) (one))) are representations of 1; (diff (diff (one) (one)) (one)) is a representation of –1.

1. Show that every number has infinitely many representations in this system.
2. Turn this representation of the integers into an implementation by writing zero, is-zero?, successor, and predecessor, as specified on page 32, except that now the negative integers are also represented. Your procedures should take as input any of the multiple legal representations of an integer in this scheme. For example, if your successor procedure is given any of the infinitely many legal representations of 1, it should produce one of the legal representations of 2. It is permissible for different legal representations of 1 to yield different legal representations of 2.
3. Write a procedure diff-tree-plus that does addition in this representation. Your procedure should be optimized for the diff-tree representation, and should do its work in a constant amount of time (independent of the size of its inputs). In particular, it should not be recursive.

2.2 Representation Strategies for Data Types

When data abstraction is used, programs have the property of representation independence: programs are independent of the particular representation used to implement an abstract data type. It is then possible to change the representation by redefining the small number of procedures belonging to the interface. We frequently rely on this property in later chapters.

In this section we introduce some strategies for representing data types. We illustrate these choices using a data type of *environments*. An environment associates a value with each element of a finite set of variables. An environment may be used to associate variables with their values in a programming language implementation. A compiler may also use an environment to associate each variable name with information about that variable.

Variables may be represented in any way we please, so long as we can check two variables for equality. We choose to represent variables using Scheme symbols, but in a language without a symbol data type, variables could be represented by strings, by references into a hash table, or even by numbers (see section 3.6).

2.2.1 The Environment Interface

An environment is a function whose domain is a finite set of variables, and whose range is the set of all Scheme values. Since we adopt the usual mathematical convention that a finite function is a finite set of ordered pairs, then we need to represent all sets of the form {(var_1, val_1), … , (var_n, val_n)} where the var_i are distinct variables and the val_i are any Scheme values. We sometimes call the value of the variable *var* in an environment *env* its *binding* in *env*.

The interface to this data type has three procedures, specified as follows:

$$\begin{aligned}
&\texttt{(empty-env)} && = \lceil \theta \rceil \\
&\texttt{(apply-env } \lceil f \rceil \texttt{ } var\texttt{)} && = f\ (var) \\
&\texttt{(extend-env } var\texttt{ } v\texttt{ } \lceil f \rceil\texttt{)} && = \lceil g \rceil, \\
&\text{where } g(var_1) = \begin{cases} v & \text{if } var_1 = var \\ f(var_1) & \text{otherwise} \end{cases}
\end{aligned}$$

The procedure empty-env, applied to no arguments, must produce a representation of the empty environment; apply-env applies a representation of an environment to a variable and (extend-env *var val env*) produces a new environment that behaves like *env*, except that its value at variable *var* is *val*. For example, the expression

```
> (define e
    (extend-env 'd 6
      (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
            (empty-env))))))
```

defines an environment *e* such that *e*(d) = 6, *e*(x) = 7, *e*(y) = 8, and *e* is undefined on any other variables. This is, of course,

only one of many different ways of building this environment. For instance, in the example above the binding of y to 14 is overridden by its later binding to 8.

As in the previous example, we can divide the procedures of the interface into constructors and observers. In this example, empty-env and extend-env are the constructors, and apply-env is the only observer.

Exercise 2.4 [**] Consider the data type of *stacks* of values, with an interface consisting of the procedures empty-stack, push, pop, top, and empty-stack?. Write a specification for these operations in the style of the example above. Which operations are constructors and which are observers?

2.2.2 Data Structure Representation

We can obtain a representation of environments by observing that every environment can be built by starting with the empty environment and applying extend-env n times, for some $n \geq 0$, e.g.,

```
(extend-env var_n val_n
   ...
   (extend-env var_1 val_1
     (empty-env))...)
```

So every environment can be built by an expression in the following grammar:

```
Env-exp ::= (empty-env)
            ::= (extend-env Identifier Scheme-value Env-exp)
```

We could represent environments using the same grammar to describe a set of lists. This would give the implementation shown in figure 2.1. The procedure apply-env looks at the data structure env representing an environment, determines what kind of environment it represents, and does the right thing. If it represents the empty environment, then an error is reported. If it represents an environment built by extend-env, then it checks to see if the variable it is looking for is the same as the one bound in the environment. If it is, then the saved value is returned. Otherwise, the variable is looked up in the saved environment.

```
Env = (empty-env) | (extend-env Var SchemeVal Env)
Var = Sym

empty-env : () → Env
(define empty-env
  (lambda () (list 'empty-env)))

extend-env : Var × SchemeVal × Env → Env
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))

apply-env : Env × Var → SchemeVal
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
       (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
       (let ((saved-var (cadr env))
             (saved-val (caddr env))
             (saved-env (cadddr env)))
         (if (eqv? search-var saved-var)
           saved-val
           (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "No binding for ~s" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "Bad environment: ~s" env)))
```

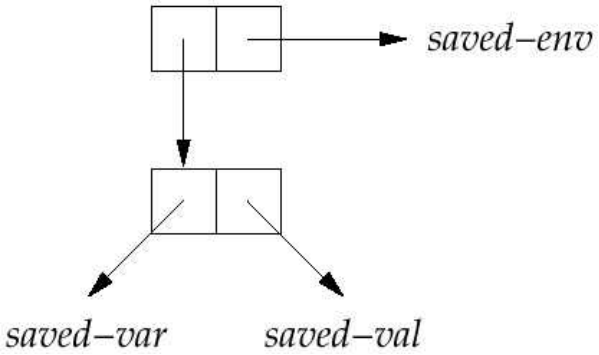
Figure 2.1: A data-structure representation of environments

This is a very common pattern of code. We call it the *interpreter recipe*:

The Interpreter Recipe

1. Look at a piece of data.
2. Decide what kind of data it represents.
3. Extract the components of the datum and do the right thing with them.

Exercise 2.5 [*] We can use any data structure for representing environments, if we can distinguish empty environments from non-empty ones, and in which one can extract the pieces of a non-empty environment. Implement environments using a representation in which the empty environment is represented as the empty list, and in which extend-env builds an environment that looks like



This is called an *a-list* or *association-list* representation.

Exercise 2.6 [*] Invent at least three different representations of the environment interface and implement them.

Exercise 2.7 [*] Rewrite apply-env in figure 2.1 to give a more informative error message.

Exercise 2.8 [*] Add to the environment interface an observer called empty-env? and implement it using the a-list representation.

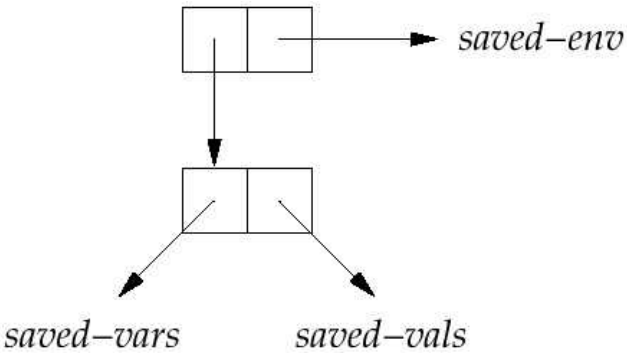
Exercise 2.9 [*] Add to the environment interface an observer called has-binding? that takes an environment *env* and a variable *s* and tests to see if *s* has an associated value in *env*. Implement it using the a-list representation.

Exercise 2.10 [*] Add to the environment interface a constructor extend-env*, and implement it using the a-list representation. This constructor takes a list of variables, a list of values of the same length, and an environment, and is specified by

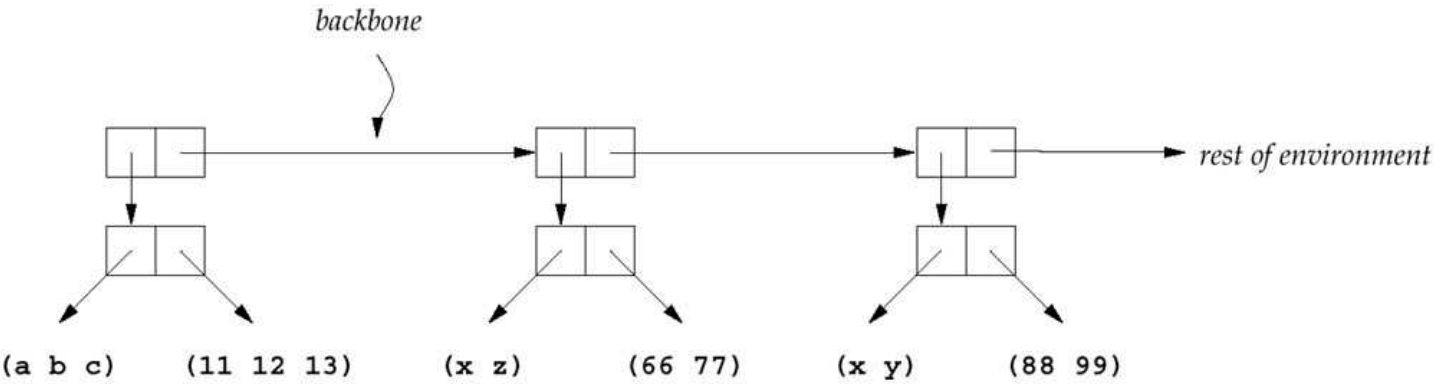
$$(\texttt{extend-env*}\ \ (var_1\ \ \ldots\ \ var_k)\ \ \ (val_1\ \ \ldots\ \ val_k)\ \ \lceil f \rceil) = \lceil g \rceil,$$

$$\text{where } g(var) = \begin{cases} val_i & \text{if } var = var_i \text{ for some } i \text{ such that } 1 \leq i \leq k \\ f(var) & \text{otherwise} \end{cases}$$

Exercise 2.11 [**] A naive implementation of extend-env* from the preceding exercise requires time proportional to *k* to run. It is possible to represent environments so that extend-env* requires only constant time: represent the empty environment by the empty list, and represent a non-empty environment by the data structure



Such an environment might look like



This is called the *ribcage* representation. The environment is represented as a list of pairs called *ribs*; each left rib is a list of variables and each right rib is the corresponding list of values.

Implement the environment interface, including extend-env*, in this representation.

2.2.3 Procedural Representation

The environment interface has an important property: it has exactly one observer, apply-env. This allows us to represent an environment as a Scheme procedure that takes a variable and returns its associated value.

To do this, we define empty-env and extend-env to return procedures that, when applied, do the same thing that apply-env did in the preceding section. This gives us the following implementation.

```
Env = Var → SchemeVal

empty-env : () → Env
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))

extend-env : Var × SchemeVal × Env → Env
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))

apply-env : Env × Var → SchemeVal
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

If the empty environment, created by invoking empty-env, is passed any variable whatsoever, it indicates with an error message that the given variable is not in its domain. The procedure extend-env returns a new procedure that represents the extended environment. This procedure, when passed a variable search-var, checks to see if the variable it is looking for is the same as the one bound in the environment. If it is, then the saved value is returned. Otherwise, the variable is looked up in the saved environment.

We call this a *procedural representation*, in which the data is represented by its *action under* apply-env.

The case of a data type with a single observer is less rare than one might think. For example, if the data being represented is a set of functions, then it can be represented by its action under application. In this case, we can extract the interface and the procedural representation by the following recipe:

1. Identify the lambda expressions in the client code whose evaluation yields values of the type. Create a constructor procedure for each such lambda expression. The parameters of the constructor procedure will be the free variables of the lambda expression. Replace each such lambda expression in the client code by an invocation of the corresponding constructor.

2. Define an apply- procedure like apply-env above. Identify all the places in the client code, including the bodies of the constructor procedures, where a value of the type is applied. Replace each such application by an invocation of the apply- procedure.

If these steps are carried out, the interface will consist of all the constructor procedures and the apply- procedure, and the client code will be representation-independent: it will not rely on the representation, and we will be free to substitute another implementation of the interface, such as the one we describe in section 2.2.2.

If the implementation language does not allow higher-order procedures, then one can perform the additional step of implementing the resulting interface using a data structure representation and the interpreter recipe, as in the preceding section. This process is called *defunctionalization*. The derivation of the data structure representation of environments is a simple example of defunctionalization. The relation between procedural and defunctionalized representations will be a recurring theme in this book.

Exercise 2.12 [*] Implement the stack data type of exercise 2.4 using a procedural representation.

Exercise 2.13 [**] Extend the procedural representation to implement empty-env? by representing the environment by a list of two procedures: one that returns the value associated with a variable, as before, and one that returns whether or not the environment is empty.

Exercise 2.14 [**] Extend the representation of the preceding exercise to include a third procedure that implements has-binding? (see exercise 2.9).

2.3 Interfaces for Recursive Data Types

We spent much of chapter 1 manipulating recursive data types. For example, we defined lambda-calculus expressions in definition 1.1.8 by the grammar

```
Lc-exp ::= Identifier
           ::= (lambda (Identifier)  Lc-exp)
           ::= (Lc-exp Lc-exp)
```

and we wrote procedures like occurs-free?. As we mentioned at the time, the definition of occurs-free? in section 1.2.4 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 can improve this situation by introducing an interface for lambda-calculus expressions. Our interface will have constructors and two kinds of observers: predicates and extractors.

The constructors are:

```
var-exp        : Var → Lc-exp
lambda-exp : Var × Lc-exp → Lc-exp
app-exp        : Lc-exp × Lc-exp → Lc-exp
```

The predicates are:

```
var-exp?         : Lc-exp → Bool
lambda-exp?  : Lc-exp → Bool
app-exp?       : Lc-exp → Bool
```

Finally, the extractors are

```
var-exp->var                    : Lc-exp → Var
```

```
lambda-exp->bound-var : Lc-exp → Var
lambda-exp->body      : Lc-exp → Lc-exp
app-exp->rator        : Lc-exp → Lc-exp
app-exp->rand         : Lc-exp → Lc-exp
```

Each of these extracts the corresponding portion of the lambda-calculus expression. We can now write a version of occurs-free? that depends only on the interface.

```
occurs-free? : Sym × LcExp → Bool
(define occurs-free?
  (lambda (search-var exp)
    (cond
      ((var-exp? exp) (eqv? search-var (var-exp->var exp)))
      ((lambda-exp? exp)
       (and
         (not (eqv? search-var (lambda-exp->bound-var exp)))
         (occurs-free? search-var (lambda-exp->body exp))))
      (else
        (or
          (occurs-free? search-var (app-exp->rator exp))
          (occurs-free? search-var (app-exp->rand exp)))))))
```

This works on any representation of lambda-calculus expressions, so long as they are built using these constructors.

We can write down a general recipe for designing an interface for a recursive data type:

Designing an interface for a recursive data type

1. Include one constructor for each kind of data in the data type.
2. Include one predicate for each kind of data in the data type.
3. Include one extractor for each piece of data passed to a constructor of the data type.

Exercise 2.15 [*] Implement the lambda-calculus expression interface for the representation specified by the grammar above.

Exercise 2.16 [*] Modify the implementation to use a representation in which there are no parentheses around the bound variable in a lambda expression.

Exercise 2.17 [*] Invent at least two other representations of the data type of lambda-calculus expressions and implement them.

Exercise 2.18 [*] We usually represent a sequence of values as a list. In this representation, it is easy to move from one element in a sequence to the next, but it is hard to move from one element to the preceding one without the help of context arguments. Implement non-empty bidirectional sequences of integers, as suggested by the grammar

```
NodeInSequence ::= (Int Listof(Int) Listof(Int))
```

The first list of numbers is the elements of the sequence preceding the current one, in reverse order, and the second list is the elements of the sequence after the current one. For example, (6 (5 4 3 2 1) (7 8 9)) represents the list (1 2 3 4 5 6 7 8 9), with the focus on the element 6.

In this representation, implement the procedure number->sequence, which takes a number and produces a sequence consisting of exactly that number. Also implement current-element, move-to-left, move-to-right, insert-to-left, insert-to-right, at-left-end?, and at-right-end?.

For example:

```
> (number->sequence 7)
(7 () ())
> (current-element '(6 (5 4 3 2 1) (7 8 9)))
6
> (move-to-left '(6 (5 4 3 2 1) (7 8 9)))
(5 (4 3 2 1) (6 7 8 9))
> (move-to-right '(6 (5 4 3 2 1) (7 8 9)))
(7 (6 5 4 3 2 1) (8 9))
> (insert-to-left 13 '(6 (5 4 3 2 1) (7 8 9)))
(6 (13 5 4 3 2 1) (7 8 9))
> (insert-to-right 13 '(6 (5 4 3 2 1) (7 8 9)))
(6 (5 4 3 2 1) (13 7 8 9))
```

The procedure move-to-right should fail if its argument is at the right end of the sequence, and the procedure move-to-left should fail if its argument is at the left end of the sequence.

Exercise 2.19 [*] A binary tree with empty leaves and with interior nodes labeled with integers could be represented using the grammar

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

In this representation, implement the procedure number->bintree, which takes a number and produces a binary tree consisting of a single node containing that number. Also implement current-element, move-to-left-son, move-to-right-son, at-leaf?, insert-to-left, and insert-to-right. For example,

```
> (number->bintree 13)
(13 () ())
> (define t1 (insert-to-right 14
               (insert-to-left 12
                 (number->bintree 13)))
>t1
(13
  (12 () ())
  (14 () ()))
> (move-to-left t1)
(12 () ())
> (current-element (move-to-left t1))
12
> (at-leaf? (move-to-right (move-to-left t1)))
#t
> (insert-to-left 15 t1)
(13
  (15
    (12 () ())
    ())
  (14 () ()))
```

Exercise 2.20 [***] In the representation of binary trees in exercise 2.19 it is easy to move from a parent node to one of its sons, but it is impossible to move from a son to its parent without the help of context arguments. Extend the representation of lists in exercise 2.18 to represent nodes in a binary tree. As a hint, consider representing the portion of the tree above the current node by a reversed list, as in exercise 2.18.

In this representation, implement the procedures from exercise 2.19. Also implement move-up, at-root?, and at-leaf?.

2.4 A Tool for Defining Recursive Data Types

For complicated data types, applying the recipe for constructing an interface can quickly become tedious. In this section, we introduce a tool for automatically constructing and implementing such interfaces in Scheme. The interfaces constructed by this tool will be similar, but not identical, to the interface constructed in the preceding section.

Consider again the data type of lambda-calculus expressions, as discussed in the preceding section. We can implement an interface for lambda-calculus expressions by writing

```
(define-datatype lc-exp lc-exp?
  (var-exp
    (var identifier?))
  (lambda-exp
    (bound-var identifier?)
    (body lc-exp?))
  (app-exp
    (rator lc-exp?)
    (rand lc-exp?)))
```

Here the names var-exp, var, bound-var, app-exp, rator, and rand abbreviate *variable expression*, *variable*, *bound variable*, *application expression*, *operator*, and *operand*, respectively.

This expression declares three constructors, var-exp, lambda-exp, and app-exp, and a single predicate lc-exp?. The three constructors check their arguments with the predicates identifier? and lc-exp? to make sure that the arguments are valid, so if an lc-exp is constructed using only these constructors, we can be certain that it and all its subexpressions are legal lc-exps. This allows us to ignore many checks while processing lambda expressions.

In place of the various predicates and extractors, we use the form cases to determine the variant to which an object of a data

type belongs, and to extract its components. To illustrate this form, we can rewrite occurs-free? (page 43) using the data type lc-exp:

```
occurs-free? : Sym × LcExp → Bool
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp (var) (eqv? var search-var))
      (lambda-exp (bound-var body)
        (and
          (not (eqv? search-var bound-var))
          (occurs-free? search-var body)))
      (app-exp (rator rand)
        (or
          (occurs-free? search-var rator)
          (occurs-free? search-var rand))))))
```

To see how this works, assume that exp is a lambda-calculus expression that was built by app-exp. For this value of exp, the app-exp case would be selected, rator and rand would be bound to the two subexpressions, and the expression

```
(or
  (occurs-free? search-var rator)
  (occurs-free? search-var rand))
```

would be evaluated, just as if we had written

```
(if (app-exp? exp)
  (let ((rator (app-exp->rator exp))
        (rand (app-exp->rand exp)))
    (or
      (occurs-free? search-var rator)
      (occurs-free? search-var rand)))
...)
```

The recursive calls to occurs-free? work similarly to finish the calculation.

In general, a define-datatype declaration has the form

```
(define-datatype type-name type-predicate-name
   {(variant-name   {(field-name predicate)}*)}+)
```

This creates a data type, named *type-name*, with some *variants*. Each variant has a variant-name and zero or more fields, each with its own field-name and associated predicate. No two types may have the same name and no two variants, even those belonging to different types, may have the same name. Also, type names cannot be used as variant names. Each field predicate must be a Scheme predicate.

For each variant, a new constructor procedure is created that is used to create data values belonging to that variant. These procedures are named after their variants. If there are *n* fields in a variant, its constructor takes *n* arguments, tests each of them with its associated predicate, and returns a new value of the given variant with the *i*-th field containing the *i*-th argument value.

The *type-predicate-name* is bound to a predicate. This predicate determines if its argument is a value belonging to the named type.

A record can be defined as a data type with a single variant. To distinguish data types with only one variant, we use a naming convention. When there is a single variant, we name the constructor a-*type-name* or an-*type-name*; otherwise, the constructors have names like *variant-name-type-name*.

Data types built by define-datatype may be mutually recursive. For example, consider the grammar for s-lists from section 1.1:

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

The data in an s-list could be represented by the data type s-list defined by

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list
    (first s-exp?)
```

```
    (rest s-list?)))

(define-datatype s-exp s-exp?
  (symbol-s-exp
    (sym symbol?))
  (s-list-s-exp
    (slst s-list?)))
```

The data type s-list gives its own representation of lists by using (empty-s-list) and non-empty-s-list in place of () and cons; if we wanted to specify that Scheme lists be used instead, we could have written

```
(define-datatype s-list s-list?
  (an-s-list
    (sexps (list-of s-exp?))))

(define list-of
  (lambda (pred)
    (lambda (val)
      (or (null? val)
        (and (pair? val)
          (pred (car val))
            ((list-of pred) (cdr val)))))))
```

Here (list-of *pred*) builds a predicate that tests to see if its argument is a list, and that each of its elements satisfies *pred*.

The general syntax of cases is

```
(cases type-name expression
   {(variant-name ({field-name}*) consequent)}*
   (else default))
```

The form specifies the type, the expression yielding the value to be examined, and a sequence of clauses. Each clause is labeled with the name of a variant of the given type and the names of its fields. The else clause is optional. First, *expression* is evaluated, resulting in some value *v* of *type-name*. If *v* is a variant of *variant-name*, then the corresponding clause is selected. Each of the *field-names* is bound to the value of the corresponding field of *v*. Then the *consequent* is evaluated within the scope of these bindings and its value returned. If *v* is not one of the variants, and an else clause has been specified, *default* is evaluated and its value returned. If there is no else clause, then there must be a clause for *every* variant of that data type.

The form cases binds its variables positionally: the *i*-th variable is bound to the value in the *i*-th field. So we could just as well have written

```
(app-exp (exp1 exp2)
  (or
    (occurs-free? search-var exp1)
    (occurs-free? search-var exp2)))
```

instead of

```
(app-exp (rator rand)
  (or
    (occurs-free? search-var rator)
    (occurs-free? search-var rand)))
```

The forms define-datatype and cases provide a convenient way of defining an inductive data type, but it is not the only way. Depending on the application, it may be valuable to use a special-purpose representation that is more compact or efficient, taking advantage of special properties of the data. These advantages are gained at the expense of having to write the procedures in the interface by hand.

The form define-datatype is an example of a *domain-specific language*. A domain-specific language is a small language for describing a single task among a small, well-defined set of tasks. In this case, the task was defining a recursive data type. Such a language may lie inside a general-purpose language, as define-datatype does, or it may be a standalone language with its own set of tools. In general, one constructs such a language by identifying the possible variations in the set of tasks, and then designing a language that describes those variations. This is often a very useful strategy.

Exercise 2.21 [*] Implement the data type of environments, as in section 2.2.2, using define-datatype. Then include has-binding? of exercise 2.9.

Exercise 2.22 [*] Using define-datatype, implement the stack data type of exercise 2.4.

Exercise 2.23 [*] The definition of lc-exp ignores the condition in definition 1.1.8 that says "*Identifier* is any symbol other than lambda." Modify the definition of identifier? to capture this condition. As a hint, remember that any predicate can be used in define-datatype, even ones you define.

Exercise 2.24 [*] Here is a definition of binary trees using define-datatype.

```
(define-datatype bintree bintree?
  (leaf-node
    (num integer?))
  (interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)))
```

Implement a bintree-to-list procedure for binary trees, so that (bintree-to-list (interior-node 'a (leaf-node 3) (leaf-node 4))) returns the list

```
(interior-node
  a
  (leaf-node 3)
  (leaf-node 4))
```

Exercise 2.25 [**] Use cases to write max-interior, which takes a binary tree of integers (as in the preceding exercise) with at least one interior node and returns the symbol associated with an interior node with a maximal leaf sum.

```
> (define tree-1
    (interior-node 'foo (leaf-node 2) (leaf-node 3)))
> (define tree-2
    (interior-node 'bar (leaf-node -1) tree-1))
> (define tree-3
    (interior-node 'baz tree-2 (leaf-node 1)))
> (max-interior tree-2)
foo
> (max-interior tree-3)
baz
```

The last invocation of max-interior might also have returned foo, since both the foo and baz nodes have a leaf sum of 5.

Exercise 2.26 [**] Here is another version of exercise 1.33. Consider a set of trees given by the following grammar:

```
Red-blue-tree      ::= Red-blue-subtree
Red-blue-subtree ::= (red-node Red-blue-subtree Red-blue-subtree)
                         ::= (blue-node {Red-blue-subtree}*)
                         ::= (leaf-node Int)
```

Write an equivalent definition using define-datatype, and use the resulting interface to write a procedure that takes a tree and builds a tree of the same shape, except that each leaf node is replaced by a leaf node that contains the number of red nodes on the path between it and the root.

2.5 Abstract Syntax and Its Representation

A grammar usually specifies a particular representation of an inductive data type: one that uses the strings or values generated by the grammar. Such a representation is called *concrete syntax*, or *external* representation.

Consider, for example, the set of lambda-calculus expressions defined in definition 1.1.8. This gives a concrete syntax for lambda-calculus expressions. We might have used some other concrete syntax for lambda-calculus expressions. For example, we could have written

```
Lc-exp ::= Identifier
          ::= proc Identifier => Lc-exp
          ::= Lc-exp(Lc-exp)
```

to define lambda-calculus expressions as a different set of strings.

In order to process such data, we need to convert it to an *internal* representation. The define-datatype form provides a convenient way of defining such an internal representation. We call this *abstract syntax*. In the abstract syntax, terminals such as parentheses need not be stored, because they convey no information. On the other hand, we want to make sure that the data structure allows us to determine what kind of lambda-calculus expression it represents, and to extract its components. The data type lc-exp on page 46 allows us to do both of these things easily.

It is convenient to visualize the internal representation as an *abstract syntax tree.* Figure 2.2 shows the abstract syntax tree of the lambda-calculus expression (lambda (x) (f (f x))), using the data type lc-exp. Each internal node of the tree is labeled with the associated production name. Edges are labeled with the name of the corresponding nonterminal occurrence. Leaves correspond to terminal strings.

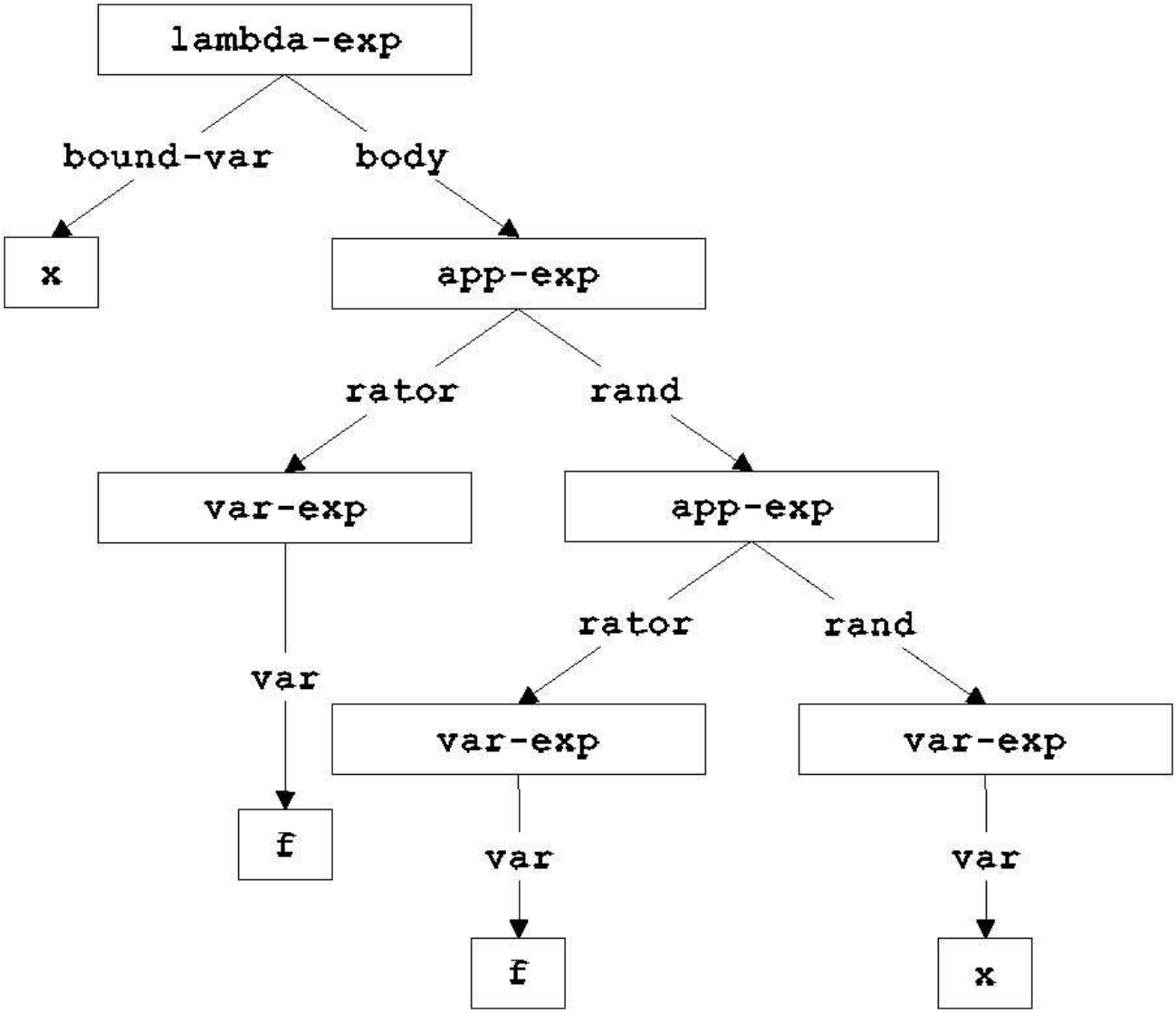


Figure 2.2: Abstract syntax tree for `(lambda (x) (f (f x)))`

To create an abstract syntax for a given concrete syntax, we must name each production of the concrete syntax and each occurrence of a nonterminal in each production. It is straightforward to generate define-datatype declarations for the abstract syntax. We create one define-datatype for each nonterminal, with one variant for each production.

We can summarize the choices we have made in figure 2.2 using the following concise notation:

Lc-exp ::= *Identifier*
`var-exp (var)`

::= `(lambda (`*Identifier*`) `*Lc-exp*`)`
`lambda-exp (bound-var body)`

::= `(`*Lc-exp* *Lc-exp*`)`
`app-exp (rator rand)`

Such notation, which specifies both concrete and abstract syntax, is used throughout this book.

Having made the distinction between concrete syntax, which is primarily useful for humans, and abstract syntax, which is primarily useful for computers, we now consider how to convert from one syntax to the other.

If the concrete syntax is a set of strings of characters, it may be a complex undertaking to derive the corresponding abstract syntax tree. This task is called *parsing* and is performed by a *parser*. Because writing a parser is difficult in general, it is best performed by a tool called a *parser generator*. A parser generator takes as input a grammar and produces a parser. Since the grammars are processed by a tool, they must be written in some machine-readable language: a domain-specific language for writing grammars. There are many parser generators available.

If the concrete syntax is given as a set of lists, the parsing process is considerably simplified. For example, the grammar for lambda-calculus expressions at the beginning of this section specified a set of lists, as did the grammar for define-datatype on page 47. In this case, the Scheme read routine automatically parses strings into lists and symbols. It is then easier to parse these list structures into abstract syntax trees as in parse-expression.

```
parse-expression : SchemeVal → LcExp
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
         (lambda-exp
           (car (cadr datum))
           (parse-expression (caddr datum)))
         (app-exp
           (parse-expression (car datum))
           (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

It is usually straightforward to convert an abstract syntax tree back to a list-and-symbol representation. If we do this, the Scheme print routines will then display it in a list-based concrete syntax. This is performed by unparse-lc-exp:

```
unparse-lc-exp : LcExp → SchemeVal
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp (bound-var body)
        (list 'lambda (list bound-var)
          (unparse-lc-exp body)))
      (app-exp (rator rand)
        (list
                 (unparse-lc-exp rator) (unparse-lc-exp rand))))))
```

Exercise 2.27 [*] Draw the abstract syntax tree for the lambda calculus expressions

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

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

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

Exercise 2.28 [*] Write an unparser that converts the abstract syntax of an lc-exp into a string that matches the second grammar in this section (page 52).

Exercise 2.29 [*] Where a Kleene star or plus (page 7) is used in concrete syntax, it is most convenient to use a *list* of associated subtrees when constructing an abstract syntax tree. For example, if the grammar for lambda-calculus expressions had been

$$\textit{Lc-exp} ::= \textit{Identifier}$$
$$\boxed{\texttt{var-exp (var)}}$$
$$::= \texttt{(lambda (}\{\textit{Identifier}\}^*\texttt{)}\ \textit{Lc-exp}\texttt{)}$$
$$\boxed{\texttt{lambda-exp (bound-vars body)}}$$
$$::= \texttt{(}\textit{Lc-exp}\ \{\textit{Lc-exp}\}^*\texttt{)}$$
$$\boxed{\texttt{app-exp (rator rands)}}$$

then the predicate for the bound-vars field could be (list-of identifier?), and the predicate for the rands field could be (list-of lc-exp?). Write a define-datatype and a parser for this grammar that works in this way.

Exercise 2.30 [**] The procedure parse-expression as defined above is fragile: it does not detect several possible syntactic errors, such as (a b c), and aborts with inappropriate error messages for other expressions, such as (lambda). Modify it so that it is robust, accepting any s-exp and issuing an appropriate error message if the s-exp does not represent a lambda-calculus expression.

Exercise 2.31 [**] Sometimes it is useful to specify a concrete syntax as a sequence of symbols and integers, surrounded by parentheses. For example, one might define the set of *prefix lists* by

```
Prefix-list   ::= (Prefix-exp)
Prefix-exp  ::= Int
              ::= - Prefix-exp Prefix-exp
```

so that (- - 3 2 - 4 - 1 2 7) is a legal prefix list. This is sometimes called *Polish prefix notation*, after its inventor, Jan Łukasiewicz. Write a parser to convert a prefix-list to the abstract syntax

```
(define-datatype prefix-exp prefix-exp?
  (const-exp
    (num integer?))
  (diff-exp
    (operand1 prefix-exp?)
    (operand2 prefix-exp?)))
```

so that the example above produces the same abstract syntax tree as the sequence of constructors

```
(diff-exp
  (diff-exp
    (const-exp 3)
    (const-exp 2))
  (diff-exp
    (const-exp 4)
    (diff-exp
      (const-exp 12)
      (const-exp 7))))
```

As a hint, consider writing a procedure that takes a list and produces a prefix-exp and the list of leftover list elements.