Higher Order Programming

March 5, 1987

Contents

1 Introduction 1
- 1.1 Purpose of the book . . . 1
- 1.2 A view of algorithms . . . 2
- 1.3 The role of the Scheme language . . . 3

2 Programming with Immutable Objects 4
- 2.1 Model of computation: the frame machine . . . 4
  - 2.1.1 Addressing . . . 5
  - 2.1.2 Procedures and procedure activations . . . 5
  - 2.1.3 Computational states . . . 5
  - 2.1.4 Execution . . . 6
  - 2.1.5 Continuations . . . 6
  - 2.1.6 Conditional behavior and looping . . . 6
  - 2.1.7 More on procedures . . . 7
  - 2.1.8 More on memory operations . . . 7
  - 2.1.9 A classification of objects . . . 8
  - 2.1.10 Extent of cells and objects . . . 9
- 2.2 The Scheme Language . . . 10
  - 2.2.1 Denotations for cells: Identifiers . . . 10
  - 2.2.2 Denotations for objects . . . 11
  - 2.2.3 Procedure applications . . . 11
  - 2.2.4 Special forms for binding and referring to cells . . . 13
  - 2.2.5 Lambda notation . . . 14
  - 2.2.6 Let and let* expressions . . . 18
  - 2.2.7 Lexical scoping and unlimited extent . . . 20
  - 2.2.8 Examples of higher order programming: Combinators . . . 21

2.2.9 Control forms 23
2.2.10 Quote 25
2.2.11 Other special forms 25
2.3 Procedural Schemata 27
2.3.1 Combinators 27
2.3.2 Conditionals 31
2.3.3 Primitive Recursion 32
2.3.4 Tail Recursion 34
2.3.5 Transforming Primitive Recursion to Tail Recursion 36
2.3.6 Exponentiation—An Example of an Operator 40
2.3.7 Factorization - Another Example of a Maker 43
2.3.8 Primality Testing 45

Chapter 1

Introduction

1.1 Purpose of the book

Higher order programming is a style of programming which uses functions that operate on functions (functional forms or operators), but does not avoid using destructive operations (in contrast to functional programming which disallows destructive operations). This style of programming allows the development of programs that are not just competitive with traditionally designed programs, but quite often outperform them. It also produces very concise, understandable, and highly reliable code.

Our notion of higher order programming combines several important programming paradigms developed in recent years, such as functional progrmming, object oriented programming, and abstract data types. It is our view that none of these paradigms is satisfactory by itself, but that they do complement each other. Functional programming provides us with the idea of functional forms, but we find that functional programming ideas can be extended to include non-applicative functions and functional forms. These non-applicative forms do not just increase efficiency of code, but very often allow expression of algorithms that are not easily expressible in a purely applicative style.

A similar point can be made about object oriented programming. Message passing and data encapsulations are extremely useful, but we don't want to restrict our world view by viewing everything as a message receiving object. With regard to abstract data types, the advantages they provide in permitting abstracting away from "how to" to "what" should be com-

plemented by a notion of algorithmic abstraction that permits describing "how to" independently of what kind of data is involved.

A second point is that we believe that it would be very difficult to devise a set of operators *a priori*, so that this set will be able to express precisely many complex algorithms. On the contrary we are currently engaged in an effort to extract higher order primitives from large classes of algorithms and data structures, such as sorting and searching algorithms, network algorithms and theorem proving algorithms. We advocate an minimal approach, in that no operator is introduced unless there is a real algorithm that requires it.

A third point is that we view the science of programming as primarily a practical science. While we are willing to sacrifice some efficiency for clarity, we are not willing to accept an algorithm description that is "cute" but grossly inefficient. It is our opinion that a lot of work in functional programming has neglected the issues of algorithmic complexity.

1.2 A view of algorithms

It is our experience that there are many important algorithms in the literature that are extremely hard to implement using conventional programming techniques. For example, it is indicative that several books on algorithm design and analysis, such as Sedgewick [ ], Tarjan [ ], and Gonnet [ ], mention binomial queues as the best implementation of mergeable priority queues, but do not even attempt to give an implementation. Even the original papers where binomial queues are introduced give only pseudo-code descriptions of the algorithms. Yet higher order primitives make implementing these algorithms easy.

Moreover, in our experience, the availability of higher order primitives facilitates the discovery of new algorithms and makes it feasible to study them experimentally.

In this paper we use a functional form called "reduction" to derive a data structure called a tournament queue, similar to binomial queues, and a family of sorting algorithms based on it. The full discussion of the complexity of these algorithms can be found in []. We then introduce encapsulations, which are collections of data and procedures similar to clusters in CLU [], modules in Modula [], and packages in Ada []. The major distinctions be-

tween encapsulations and previous mechanisms are that encapsulations are first class objects (e.g., they can be passed as parameters, returned as as values, stored in variables) and that they are function constructing objects which receive messages and return functions. The functions that are thus obtained do not have the overhead usually associated with message passing and dispatching.

We use these encapsulations and the data structures previously derived to implement a restricted priority queue, as needed for an example application, a program for allocating a budget according to a given assignment of priorities and costs to a large number of items.

1.3 The role of the Scheme language

Our approach is not dependent on a particular language, but we find it most convenient to work in a language in which functions are first class objects. The Scheme language [], a modern dialect of Lisp, turns out to be almost ideal in this respect. Other Lisp dialects such as Common Lisp can be used, though less conveniently, and higher order programming techniques have been used by our students in other widely used languages such as C (these techniques have been taught in a graduate level course at Polytechnic University). In this paper we use a subset of Scheme which we briefly describe in the following section. Any reader who wishes to experiment with the example methods described can do so using any of the several widely available implementations of Scheme.

Chapter 2

Programming with Immutable Objects

2.1 Model of computation: the frame machine

We will begin the description of our approach to programming by describing an ideal computer for higher order programming. This "virtual machine" is actually very close in structure to existing hardware/software implementations of the Scheme programming language, but presenting it in a somewhat idealized form allows us to concentrate on the main issues needed for understanding the programming techniques and principles presented in later chapters.

Our ideal computer is capable of storing and manipulating *objects* of several different types, including not only numbers and characters but also procedures, which provide the machine with instructions and with organization of its memory. Before considering procedures and other object types, let us examine the structure of the machine's memory.

The memory of the frame machine consists of *cells*, each of which can hold any object. The memory is organized into

- *frames:* finite sequences of cells,

and the frames are further organized into

- *environments:* finite sequences of frames.

Because of the central role played by frames, we will call our ideal computer "the frame machine."

2.1.1 Addressing

Environments control which cells can be addressed by the frame machine's instructions. The addresses in all instructions are of the form (i, j), in which i is a frame number and j is the number of a cell within that frame. Such an address is taken to refer to a cell in a "current environment," to be defined in a moment, and thus it is not possible for instructions to refer to cells outside the current environment.

We assume the cost of accessing cell (i, j) varies between a best case which is a constant time and a worst case which is proportional to $i + 1$. (As we shall see later, i is always small, hence memory access is essentially bounded by a constant.)

2.1.2 Procedures and procedure activations

The frame machine gets its instructions and environments from procedure objects. A procedure object consists of an instruction sequence and an environment. From a procedure the machine is capable of creating a *procedure activation*, which also consists of an instruction sequence, called its *saved instruction sequence*, and an environment.

2.1.3 Computational states

The machine operates on a *current state* (of computation), consisting of

- an instruction sequence,
- procedure activation sequence.

These two parts are called the *current instruction sequence* and the *current procedure activation sequence*, and the first members of these sequences are called the *current instruction* and the *current procedure activation*; the environment part of the latter is called the *current environment*. See Figure ??.

2.1.4 Execution

The frame machine proceeds by repeatedly removing the current instruction from the current instruction sequence and carrying it out. The machine keeps pulling off instructions until the current instruction sequence is exhausted, then it makes the saved instruction sequence in the current procedure activation be the new current instruction sequence and removes the current procedure activation. It proceeds in this way until the entire procedure activation sequence is exhausted, at which point it stops.

However, the machine usually will not just march straight through all of the original procedure activations and stop, because there are instructions, called *procedure application* instructions, that are capable of creating a new procedure activation and putting it in front of the existing sequence. This new procedure activation has

- as its environment: an environment obtained from the procedure
- as its saved instruction sequence: the current instruction sequence (those instructions following the procedure application instruction)

The new current instruction sequence is the instruction sequence obtained from the procedure. After this is obtained, execution continues as described above.

2.1.5 Continuations

There are also instructions for saving the entire current state as an object that can later be used to replace whatever current state then exists. Because of the existence of these "continuation instructions," the procedure activations that are removed as the computation proceeds are not always just discarded; they may later be reused as parts of a continuation.

2.1.6 Conditional behavior and looping

Some instructions cause the frame machine to skip over one or more later instructions in the current intruction sequence if some condition if satisfied, e.g., if some cell contains a certain object. This kind of conditional behavior combines with procedure application instructions to give the frame machine the same capability of branching and looping as in other general purpose

computing machines: although there is no backward branching within the instruction sequence of a single procedure, one can always achieve the same effect by breaking down the procedure into smaller ones.

2.1.7 More on procedures

Procedures can be applied to some fixed number of objects (inputs), possibly having an effect, and returning some object as the result (output) of the application (except that some procedures, for some inputs and environments, may never terminate execution). By an effect is meant the creation of frames or objects, or assignment of new values to the cells in the created or already existing frames within the current environment or to components of objects. The effect of an application of procedure p is achieved by executing its instruction sequence, whose instructions direct the machine to carry out operations on cells and objects directly or indirectly via procedure applications, possibly including p itself (recursive applications). Some of the computations may occur simultaneously, but such parallel execution is not a requirement of a Scheme implementation. What series of computations are carried out can depend not only on the procedure inputs but also on the values stored in cells in the current environment.

The number of input objects for application of the procedure is fixed[1] and is determined at the time the procedure is created. Creation of a procedure is also based on a description of the series of computations to be carried out, which is embodied in the procedure as an instruction sequence. It is the central purpose of the Scheme programming language, as discussed in Chapter 3, to provide a notation for writing such computational descriptions.

2.1.8 More on memory operations

Cells can be created, assigned to and retrieved from; the value that is thus stored and retrieved can be any object. We use the term "accessing" for the operations of assigning a value to a cell or retrieving a value from it. It is helpful to think of a cell as a box whose contents is an object, as in Figure ??.

[1] In Chapter 3, we shall see how the Scheme language provides an exception to this rule.

Actually, what is stored in the cell is in most cases a *pointer* to the object, rather than the object itself; see Figure ??. Pointers are a type of object that have only one operation defined: that of checking for identity of two pointers. Pointers are also called addresses (but should not be confused with the (i, j) addresses of cells). A pointer to an object is stored in a cell unless the object itself is "small," in the sense that it will fit into the same storage as ordinarily would be used to hold a pointer. As a consequence, the time it takes to access a cell is essentially independent of the type of object.

Frames consist of a fixed number of cells, which are created at the same time as the frame. It is possible to have a frame with no cells in it, called an empty frame.

An environment is a nonempty sequence of frames, the last member of which is called the *global environment*. A new environment is always created from a newly created frame and an existing environment. The new frame is the first member, f_0, of the sequence of frames in the new environment and the members of the existing environment become $f_1, f_2, \ldots$, etc.

2.1.9 A classification of objects

The types of object that the frame machine can manipulate can be divided into three groups:

- *immutable, simple objects*, or constants, including numbers, symbols, characters, the empty list, ports (objects that provide for input and output), and a few others;
- *mutable, structured objects*, including lists, vectors, and strings, which have parts that are subject to change over time;
- *procedural objects*, including procedures and continuations.

An important distinction exists between these groups in terms of tests for *operational equivalence* of objects. Two objects are operationally equivalent if and only if there is no computation that will distinguish them, other than by using operations that detect a difference in the pointers (if any) that are associated with the objects.

- With simple objects, there are machine operations that check for operational equivalence of two objects of the same type. For numbers, for example, there is an operation = that provides this check. These checks are typically computable in constant time, with some exceptions: for numbers, for example, in implementations in which they can be arbitrarily many digits long, the time will depend on the length of the shortest operand.

- With structural objects, the situation is more complicated. For lists, for example, there is an operation **equal** that is only an approximation to operation equivalence, since it may never terminate for some inputs (*circular lists*). However, such an operation can be programmed, as will be shown in Chapter ??.

- With procedural objects, there is no operation provided that even approximates operational equivalence; and according to results of computability theory it is not even possible to program such an operation.

Further discussion of operational equivalence will be given in Chapter ??.

At this point we shall not give a detailed discussion of the different object types. Most of the details will emerge in later chapters, or can be found in the *Revised*^{3} *Report on the Algorithmic Language Scheme*, which is reprinted in Appendix ??.

2.1.10 Extent of cells and objects

One other point about the frame machine model of computation is crucial to effective use of Scheme (or any Lisp dialect). The frame machine provides means of creating arbitrary numbers of cells and of objects of each object type, but provides no way of destroying them. Cells and objects are said to have *unlimited extent*. This amounts to assuming that memory is infinite. Since memory is of course limited in any real implementation, it might seem that the lack of any facility for destroying cells and objects would severly restrict the programmer, requiring great care to avoid creating cells and objects; for example, one might create lists or vectors only when none of the existing ones were available to be reused. For programs which must process large amounts of data, this strategy would lead to considerable programming difficulties.

Fortunately, Scheme and all other Lisp dialects are supported by implementations that can recognize, in most cases, the situtation that a cell or object cannot possibly matter to any future computation, in which case the memory it occupies can be reclaimed. This reclamation is called "garbage collection," a process that occurs when a program invokes an operation that tries to create an object but no memory is available for it to occupy. Garbage collection can be regarded as a means of mapping infinite memory into finite space (provided the amount of space actually needed at any given time is within the space available). The Scheme or Lisp programmer can depend on this feature and be more relaxed about creating cells and objects.

Nonetheless, the programmer should not be too relaxed! Creating cells and objects takes time, often substantially more than reusing existing ones, especially when the time for garbage collection is taken into account. In this text we will put considerable stress on techniques that serve the programmer well in avoiding unnecessary creation of cells and objects.

2.2 The Scheme Language

Scheme provides a notation for defining and applying procedures. It first of all provides a means of referring to cells and of denoting each member of each of the types of objects in the above model of computation.

2.2.1 Denotations for cells: Identifiers

An *identifier* is a sequence of characters that contains no special characters and begins with a character that cannot begin a number. Also, + and - are identifiers. Identifiers are sometimes used in Scheme programs to denote symbols, but in most cases an identifier in a Scheme program denotes a cell. A cell associated with a occurrence of an identifier is called a ***binding*** of the identifier. Identifiers are used in Scheme instead of machine addresses, and in fact occurrences of identifiers in a Scheme program can be put into one to one correspondence with the (i, j) machine addresses described for the Scheme virtual machine in the previous chapter. As a consequence, an occurrence of an identifier can only refer to a cell in the current environment. Since different environments become current during the course of

computation, an occurrence of an identifier may have different bindings at different times during the computation. See Figure xx.

Also associated with an occurrence is a *region* of the program in which other occurrences of the identifier have the same binding. This region is also called the ***scope*** of the identifier. See Figure xx. Scheme is said to be it lexically scoped, since scopes are determined by how the text of program parts are nested among one another, rather than as an effect of computation (it dynamic scoping, which is what most Lisp dialect have, the other major exception being Common Lisp). The exact scope rules for Scheme will be given in later sections, as will a discussion of the significance of lexical scoping and how it interacts with the feature of unlimited extent of cells.

2.2.2 Denotations for objects

Only an overview of how the different types of objects are denoted in Scheme will be given at this point. More details will be given as other parts of the language and programming examples are discussed later.

- Numbers are denoted by sequences of digits and other characters such as period (used as a decimal point), using decimal representation.
- Symbols are denoted just by sequences of characters, though to distinguish them from identifiers in a program a quoting convention is used, as described in Section xx.
- Strings are denoted by sequences of characters enclosed in double quotation marks, e.g., `"foo"`.
- The empty list is denoted by `()`.

Denotations of pairs, vectors, ports, procedures, and continuations will be discussed in later sections.

2.2.3 Procedure applications

Each Scheme implementation comes equipped with certain procedures already defined in its initial environment, and some of these procedures provide the basic operations on the different types of Scheme objects. For example,

```
(* 3 4)
```

denotes application of a procedure that takes two numbers and computes their product. This procedure object is contained in a cell bound to the identifier * in the initial environment of the Scheme system. Similarly, + stands for an addition procedure, and

```
(+ (* 3 4) 5)
```

denotes the application of the addition procedure to the result of `(* 3 4)` and 5, which produces a result 17. The general notation for procedure application is

$$(proc\ input_1\ input_2\ \ldots\ input_n)$$

where each of *proc* and $input_1, \ldots, input_n$, is either

- a denotation of an object, or
- a denotation of a procedure application, or
- a "special form" (which will be explained in the next subsection),

such that evaluation of *proc* yields as its value a procedure object that expects n inputs.

Before going on, we should mention a very important point about Scheme systems and how the reader can best understand many of the points to be made in the following discussion. A Scheme system is *interactive*, in the sense that when activated it creates a port and connects it to the user's terminal (or to the keyboard and screen of his or her microcomputer or workstation). The user can input sequences of characters that denote Scheme objects, procedure applications, or special forms, which the Scheme system will immediately evaluate. This kind of interactive program construction is familiar to Basic programmers, but contrasts with the "batch-oriented" approach of Fortran or Pascal. The reader will find it extremely helpful to have access to a Scheme system to use to work though examples and exercises that are presented.

All lines displayed in this text that are

```
in this font
```

are legal input to the Scheme system and the reader is strongly encouraged to enter them while working through the text. In some cases, the results that are shown depend upon previous lines having been entered.

Even programmers who are already experienced in some other Lisp dialect should work through the examples and exercises on higher order programming techniques with the aid of either a Scheme system or, with some adaptation, another Lisp system. (Appendix xx discusses how to adapt the essential techniques to several other Lisp dialects.)

2.2.4 Special forms for binding and referring to cells

Cells are not Scheme objects and the Scheme notation for procedure applications does not, by itself, provide a means of operating on cells. For this purpose, one must use a class of special forms; e.g.,

```
(define x 3)
```

means create a cell, bind the identifier `x` to it, and assign the object `3` to the cell. In

```
(* x 4)
```

the occurrence of the identifier `x` denotes retrieval of the object in the cell bound to `x`. Similarly, the occurrence of `*` denotes retrieval of the object in the cell bound to `*`. If that object is the multiplication procedure, as it is in the initial Scheme environment, and the above `define` operation has just been performed, then the result of this procedure application is `12`.

It is important to understand why `define` cannot be a procedure. If `(define x 3)` were a procedure application, the occurrence of `x` would mean retrieval of an object from a cell already bound to it (provided such a cell even existed), and all that define would "see" would be the object, not identifier or the cell. For `define` to work properly, it must operate on the identifier itself to associate it with a cell.

Thus `define` is one example of a Scheme special form and just the appearance of an identifier in certain places is also considered a special form. Scheme provides these special forms to denote operations on cells and to provide ways of composing computations that cannot be provided with procedure applications alone.

`Define` does not always create a new cell; in

```
(define x 4)
```

if **x** already has a cell bound to it (in the region of this occurrence of **x**), then the new value, **4**, is just assigned to the existing cell. The special form **set!** also changes the value stored in cell, but it assumes that a cell already exists:

```
(set! x 5)
```

would be okay at this point, but

```
(set! y 5)
```

would cause an error, since no cell is bound to **y** in the initial Scheme environment.

2.2.5 Lambda notation

The basic notation for denoting a procedure is in terms of a special form called a **lambda** expression. For example,

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

denotes a procedure that takes one input, assumed to be a number, and outputs the number that is the square of its input. This procedure can be given a name, **square** (i.e., can be assigned to a value cell created and associated with **square**), using **define**:

```
(define square (lambda (z) (* z z)))
```

so that the subsequent input

```
(square 5)
```

yields the value 25. But naming the procedure isn't necessary,

```
((lambda (z) (* z z)) 5)
```

works just as well. The advantage of naming the procedure is for convenience in reusing it in other computations, principally in composing the definitions of other procedures. For example, we can

```
(define sum-of-squares
   (lambda (x y)
      (+ (square x) (square y))))
```

so that

```
(sum-of-squares 3 4)
```

yields 25. This definition of `sum-of-squares` could have preceded that of `square`, because in obtaining a value for

```
(lambda (x y)
   (+ (square x) (square y)))
```

the Scheme system does not attempt to get the procedure named by `square`; rather it produces a procedure that, when it is applied, will apply *whatever procedure is then named by* `square` to the values of `x` and `y`.

Let us examine closely how the Scheme system evaluates the application

```
(sum-of-squares (+ 1 2) 4)
```

First, each constituent of the application is evaluated:

- evaluation of this occurrence of identifier `sum-of-squares` means retrieval of the procedure named by `sum-of-squares`, in this case the procedure created by the above `(define sum-of-squares ...)`.
- evaluation of `(+ 1 2)` results in the number 3.
- evaluation of 4 results in the number 4.

Then the retrieved procedure object is applied to 3 and 4. Since the procedure is the object created by

```
(lambda (x y) (+ (square x) (square y)))
```

application of it causes two cells to be created and bound to `x` and `y`, and the values 3 and 4 are assigned to these cells. Then the form

```
(+ (square x) (square y))
```

is evaluated, which means that each of `+`, `(square x)`, and `(square y)` will be evaluated, then the procedure retrieved from `+` will be applied.

1. Evaluating `+` retrieves the addition procedure.
2. Evaluating `(square x)` causes evaluating `square` and `x`, then application of the procedure retrieved from `square`
3. Evaluating `square` retrieves the procedure created by

   `(lambda (z) (* z z))}`

4. Evaluating `x` retrieves the value `3`
5. Application of the procedure created by `(lambda (z) (* z z))` causes a new cell to be created and bound to `z` and the value `3` to be assigned to it. Then `(* z z)` is evaluated, resulting in the value `9`.
6. Similarly, evaluating `(square y)` yields the value `16`
7. Finally, the procedure named by `+` is applied to `9` and `16` to yield `25`.

One point about evaluation of `(square x)` is extremely important. In defining `square`, we could just as well have used the identifier `y`,

```
(define square (lambda (y) (* y y)))
```

since application of the procedure created by `(lambda (y) (* y y))` causes a *new* cell to be created and bound to `y`, and this cell is used only in the region of the identifier `y`, which in this case is just the `lambda` expression. Thus storing the value of `3` into this cell would not affect the value `4` stored in the cell bound to the `y` of `(lambda (x y) (square x) (square y))`.

As another example, consider

```
(define b 1)

((lambda (a b)  ;  A procedure,
     (a 5))     ;     call it procedure X
 (lambda (x) ;  A procedure, call it procedure Y,
     (+ x b))   ;     first input to procedure X
 2)             ;  Second input to procedure X
```

in which

- the binding of a is a cell containing procedure Y,
- the binding of the b in procedure X is a cell containing 2.

Thus in evaluating `(a 5)`, what cell does the `b` that occurs in `(+ x b)` refer to? I.e., is the result 6 or 7?

According to the lexical scope rules of Scheme, this `b` refers to the cell containing 1 that was created by the `define`, not to the the cell containing 2 that was created by `(lambda (a b) (a 5))`. The region of the latter binding of `b` is just `(lambda (a b) (a 5))` and thus does not affect the occurrence in `(lambda (x) (+ x b))`. Dynamic scoping, on the other hand, would give the opposite conclusion.

In general the following notation is used in Scheme to denote a procedure:

```
(lambda (var1 var2 ... varn )
   form1
   form2
   ...
   formk)
```

where var_1, var_2, ..., var_n are identifiers and $form_1$, $form_2$, ..., $form_k$ are are any Scheme expressions. This denotes a procedure that takes n values as inputs and produces an output value by

- creating n cells, binding them to var_1, var_2, ..., var_n , and storing in them the n values passed as inputs,
- successively evaluating $form_1$, $form_2$, ..., $form_k$ for their effect,
- returning the value output by $form_k$.

The occurrences of var_1, var_2, ..., var_n at the beginning of the `lambda` expression are called *binding occurrences.* A binding occurrence has two associations which are crucial to the precise definition of execution of Scheme procedures:

- the cell, call it c, that is created and bound to var_i whenever the procedure is executed.

- the region that is defined by the scope rules of the language definition. Scheme constructs that define regions for identifiers are called *binding forms*. In the case of a lambda binding form, the region of the binding occurrences is the entire lambda form.

This region defines where in the text of programs other occurrences of var_i refer to the same cell, c, according to the following

- nesting rule: every occurrence of var_i refers to the binding of the identifier that established the innermost of the regions containing the occurrence.

Thus within $form_1$, $form_2$, ..., $form_k$, occurrences of var_i refer to the cell c, unless they lie within another binding form that defines a region for var_i, such as another

(lambda (...var_i ...) ...)

See Figure xx.

Other binding forms besides `lambda` include `let` and `let*`, which we now discuss.

2.2.6 Let and let* expressions

An example of `let` is

```
(let ((x (+ a 1))
      (z (* b c)))
     (* (+ x z) x))
```

which means create new cells for `x` and `y` and assign the value of `(+ a 1)` to the cell for `x` and the value of `(* b c)` to the cell for `z`, then evaluate `(* (+ x z) x)` and return its value. The general form of `let` is

(let ((var_1 $vform_1$)
      ...
      (var_n $vform_n$))
  $form_1$
  ...
  $form_k$)

meaning evaluate $vform_1$, ..., $vform_n$in some order, assign the value of $vform_i$ to a new cell created for var_i, then successively evaluate $form_1$, $form_2$, ..., $form_k$ for their effect. The value of $form_k$ is returned as the value of the `let`. The region of each of the var_i is $form_1$, $form_2$, ..., $form_k$.

The `let` special form can be defined in terms of `lambda`: the above general form has the same meaning as

```
((lambda (var_1 ... var_n)
   form_1
   ...
   form_k)
 vform_1 ... vform_n)
```

and the example has the same meaning as

```
((lambda (x z)
  (* (+ x z) x))
  (+ a 1) (* b c))
```

Thus `let` is not strictly necessary, but is usually more readable than the corresponding expression using `lambda`.

The translation of `let` in terms of `lambda` shows that in an expression such as

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

which is equivalent to

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

the `x` in `(* x 4)` does *not* refer to the value 3 associated with `x` by the `(x 3)` part. Rather, it refers to whatever value is associated with `x` in the lexical region of which the `let` is a part (this region does not include the `lambda` form, since that form contains another occurrence of `x`).

To be able to refer to the new bindings of identifiers, one can use the `let*` special form, which has the general form

```
(let* ((var_1 vform_1)
       ...
       (var_n vform_n))
   form_1
   ...
   form_k)
```

This is equivalent to

```
(let ((var_1 vform_1))
   (let ((var_2 vform_2))
     ...
        (let((var_n vform_n))
            form_1
            ...
            form_k) ...)
```

and thus occurrences of var_i in $vform_j$, when $j > i$, mean retrieval from the cell newly bound to var_i and containing the value of $vform_i$.

2.2.7 Lexical scoping and unlimited extent

In the chapter on the Scheme model of computation, we mentioned that all cells and objects have unlimited extent, meaning that once created they are never destroyed. This feature combines with lexical scoping to give Scheme a capability that is extremely important to higher order programming.

Consider a very simple procedure that returns a procedure as its output:

```
(define make-constant-adder
   (lambda (c)
     (lambda (x) (+ x c))))

(define one-plus (make-constant-adder 1))

(define two-plus (make-constant-adder 2))

(one-plus 5)

(two-plus 5)
```

When `(one-plus 5)` is evaluated, the procedure that is applied to 5 is the procedure returned by `(make-constant-adder 1)`, namely

```
(lambda (x) (+ x c))}
```

in which `c` refers to the cell created by the application of

```
(lambda (c)
   (lambda (x) (+ x c))))
```

to 1 in `(make-constant-adder 1)`. Thus this cell contains 1, and `one-plus` is a procedure that returns its input plus 1. But `(make-constant-adder 2)` causes another binding of `c` to be created, with 2 stored in it, so that in the evaluation of `(two-plus 5)` the `c` in `(lambda (x) (+ x c))` refers to that cell.

The main point is that a binding continues to live even after termination of the procedure that created it, and different bindings of the same occurrence of an identifier, like *c*, can coexist. Without this feature, most of the higher order programming techniques that we will discuss could not be carried out in a direct and simple way.

2.2.8 Examples of higher order programming: Combinators

The `make-constant-adder` procedure is the first example we have given of a higher order procedure, i.e., one that takes a procedure as an input or produces one as its output (most higher order procedures do both, but make-constant-adder only outputs a procedure). We will frequently use the term *operator* for higher order procedures. `Make-constant-adder` is just an instance of the more general and useful operators

```
(define (bind-1-of-2 procedure constant)
  (lambda (x)
     (procedure constant x)))

(define (bind-2-of-2 procedure constant)
   (lambda (x)
      (procedure x constant)
```

that make a procedure that takes two inputs into one that takes only one input.

Problem: What does the procedure `foo` produced by

```
(define foo (bind-1-of-2 / 1)
```

do?

`Bind-1-of-2` and `bind-2-of-2` are examples of operators found in the theory of "combinators" first introduced by M. Schoenfinkel in 1924 (50 years before Scheme!). Another example is

```
(define (double-input procedure)
   (lambda (x) (procedure x x))
```

from which another way to get `square` is

```
(define square (double-input *))
```

or to get a doubling procedure is

```
(define double (double-input +))
```

Thus, as in the example of `make-constant-adder`, a single occurrence of an identifier, in this case the occurrence of `procedure` in the definition of `double-input`, has two coexisting bindings, one to a cell containing the multiplication procedure and the other to a cell containing the addition procedure.

There is an analogy between combinators and composition of digital circuits; for example, `double-input` corresponds to splitting an input to a circuit with internal wiring to become 2 inputs to an internal circuit:

Corresponding to wiring of the output of one circuit as the input to another, as in

is a combinator that composes two procedures:

```
(define (compose p q)
  (lambda (x) (p (q x))))
```

Problem: Define a combinator `make-difference` that takes two 1-input procedures `f` and `g`, assumes that they produce numbers as outputs, and outputs a procedure `d` such that `(d x) = (f x) - (g x)`.

Problem: Define an combinator `transpose` that takes a 2-input procedure `f` and returns a 2-input procedure `g` such that `(g x y) = (f y x)`.

What is `((transpose -) 5 2)`?

Problem: What does the following procedure do?

```
(define foobar
  ((transpose make-difference) identity (double-input *)))
```

Such combinators as these can be made the base of a style of programming sometimes called *variable-free programming*, because they eliminate the need to ever use binding forms directly in defining new procedures. Although some authors advocate widespread use of such a programming style, we shall not pursue that line in this text, mainly because we feel a completely combinator based style tends to be hard to read, but also because, with current compiling techniques, it tends to be significantly less efficient than more traditional programming styles. On the other hand, some of the combinators mentioned above are handy at times for quickly contructing experimental code.

2.2.9 Control forms

The next class of special forms which we shall describe are control forms, which are not binding forms; their purpose is composition of forms in a way that is not conveniently described just using procedure application. The simplest of these is the `begin` form,

```
(begin form1
       form2
       ...
       formk)
```

which means evaluate $form_1$, $form_2$, ..., $form_k$ in sequence, whereas in procedure applications no order of evaluation is implied among evaluation of inputs. The output of the `(begin ...)` is the output of $form_n$.

Two other sequencing forms, `if` and `cond`, provide for selection, among a number of forms, of one or more to be evaluated. For example,

```
(if (> foo 4) (* foo 5) (+ foo 6))
```

means test whether `foo` is greater than `4`, and if it is, compute `(* foo 5)` and return its value, otherwise compute `(+ foo 6)` and return its value. In general, the meaning of

(**if** *test* $form_1$ $form_2$)

is:

- the test is evaluated, and
- if it results in a "true" value then $form_1$ is evaluated and its value is returned as the value of the `if` form;
- if the evaluation of the test results in a "false" value then $form_2$ is evaluated and its value is returned as the value of the `if` form.

Only one of $form_1$ and $form_2$ is evaluated, which is why `if` cannot be defined as a procedure, since application of a procedure always first evaluates all of the forms specifying the inputs.

The symbol `#t` is used to represent "true" and `#f` to represent "false"; the operation `>` always returns one of these two symbols. In places like the *test* part of `if`, however, that are described as being a true or false value, any Scheme object is allowed, with the interpretation that only `#f` and the empty list, `'()`, are false; every other object is regarded as true.

For more elaborate sequences of tests, it is often more convenient to to use `cond` instead of `if`:

```
(cond (test1 form1,1 ...)
      (test2 form2,1 ...)
      ...
      (testn formn,1 ...)
      (else formn+1,1 ...))
```

is equivalent to

```
(if test1
    (begin form1,1 ...)
    (if test2
        (begin form2,1 ...)
        ...
```

```
(if test_n
    (begin form_{n,1} ...)
    (begin form_{n+1,1} ...)...)
```

2.2.10 Quote

The last special form we shall discuss is `quote`:

`(quote` *object*`)`

means take the *object* as given, without attempting to evaluate it. The abbreviation `'`*object* is permitted for `(quote` *object*`)`. For example,

```
(define foo 3)
(define bar foo)
bar
```

yields 3, but

```
(define bar 'foo)
bar
```

yields the symbol `foo`. That is, the object that is stored in the cell bound to `bar` is the symbol `foo`; there is no attempt to treat `foo` as an identifier and retrieve a value from a cell bound to it (indeed, as an identifier it might not even have a binding).

The use of `'` is similar to the use of quotation marks in denoting strings, e.g., `"foo"` denotes a string in Scheme, but `'` can be used with any type of object. It is however unnecessary to use `'` with numbers, strings, or the special symbols `#t` or `#f`.

2.2.11 Other special forms

There are several other special forms, but we defer their discussion until they are used in examples later in the text. (The special forms that have been given are the ones that are most essential to the style of programming advocated in this text. It should also be noted that some of the special forms discussed above, such as `cond`, have variants that have not been mentioned, in the interest of brevity. The reader can consult [the R3 report] for further details. One variant of `define` that we will make use of (even though it is not available in all implementations of Scheme) is

```
(define (identifier var1 var2 ... varn )
   form1
   ...
   formk)
```

which is equivalent to

```
(define identifier
(lambda (var1 var2 ... varn )
   form1
   ...
   formk)
```

We also assume, though it is not required in all Scheme implementations, that one or more (define ...) expressions can be included as the first forms of a (define ...):

```
(define (identifiervar1 var2 ... varn )
   (define (identifier1 ...)
       ...)
   (define (identifier2 ...)
       ...)
   ...
   (define (identifierm ...)
       ...)
   form1
   ...
   formk)
```

with the regions of the identifiers $identifier_1, \ldots, identifier_k$ being the entire outer **define** expression. Appendix xx shows how to do without this feature if one is using a Scheme system that does not have it.

2.3 Procedural Schemata

The procedures given as illustrations in the preceding sections are referred to as straight line procedures. There is a limit to how much we can accomplish just using such simple procedures. It is therefore useful to define procedures which take other procedures as arguments. By defining such procedures, it is possible to create more complex procedures from simpler ones and thus, ultimately, compute virtually any computable function.

There are a number of different ways to combine procedures. We will explore several of the most useful ones below. We refer to these methods of combining procedures as a procedural schemata. Since procedures are first class objects in Scheme, we can easily create procedural schemata by passing procedures as arguments to other procedures.

2.3.1 Combinators

The idea of combinators was introduced by Schonfinkel in 1924 to produce a variable-free functional calculus. Later his ideas were developed further by Haskell Curry who made this calculus into a separate branch or mathematics called Combinatory Logic.

A pure combinator is a procedure, the body of which contains only formal parameters of this procedure, other lambda expressions and their parameters and procedure applications of these parameters and lambda expressions to each other.

For example, `foo` in

```
(define foo
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (lambda (w) (((w x) (w y)) (w z)))))))
```

is a pure combinator. While

```
(define bar
  (lambda (x)
    (lambda (y)
      (lambda (z) (x (+ (y z) 1))))))
```

is not since + is not bound in bar.

Pure combinators capture a notion of connecting different functional boxes together. ...

To make the formalism more tractable, combinatory logic deals with functions of only one argument. At first, this may appear too restrictive, but there is a beautiful device called "currying" which allows us to reduce all functions to functions of one argument.

We will illustrate this device by "currying" addition. Obviously, we need some way to add two numbers, but we are allowed to have only procedures of one argument. We can solve this problem by defining

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

Plus is a function that takes a number and returns a procudure that adds this number to its argument. So to add 4 and 5 we evalute ((plus 4) 5).

We can actually abstract from this and make a procedure that "currys" an arbitrary procedure of two arguments:

```
(define curry
  (lambda (procedure)
    (lambda (x)
      (lambda (y) (procedure x y)))))
```

so we can easily obtain

```
(define times (curry *))
```

and

```
(define take-away (curry -))
```

The first combinator we introduce is called the *elementary identificator*, I,

```
(define I
  (lambda (x)
    x))
```

The next combinator we introduce is the *elementary permutator*, `C` (for converse):

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

Now a procedure f1 defined as:

```
(define f1 ((C take-away) 1))
```

is equivalent to `(lambda (x) (- x 1))`.

Next, we define the *elementary duplicator*, `W` (for ???):

```
(define W
  (lambda (f)
    (lambda (x)
      ((f x) x))))
```

Thus, the procedure `f2` defined as

```
(define f2 (w times))
```

is equivalent to

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

Next, we define the *elementary compositor*, B (for ???) as:

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

Problem: What does a function `f3` do if it is defined as

```
(define f3 ((b (w times)) ((c plus) 1)))
```

Finally, we define the *elementary cancellator*, K (for cancellator) as:

```
(define K
  (lambda (c)
    (lambda (x) c)))
```

Problem: What does a function `f4` do if it is defined as

```
(define f4 (w k))
```

Here is one more complicated combinator:

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

Problem: What does a function `f5` do if it is defined as

```
(define f5 ((s (k s)) k))
```

We can continue adding new combinators forever, but fortunately this is unnecessary. There is a marvelous property called "combinatory completeness" which is possessed by a set of just two combinators `K` and `S`. It can be formally defined as follows:

Let $X = \{x_1, \ldots, x_n\}$ be a set of Scheme constants and variables. A binary expression based on X can be defined recursively: any x_i is a binary expression, and if u and v are binary expressions so is $(u\ v)$. Then for any binary expression T based on X there is an equivalent expression T' of the form: $((((\ldots((Z)\ x_1)\ x_2)\ x_3)\ldots)\ x_n)$, where Z is a binary expression based on $\{K, S\}$.

For example if we are given an expression A:

```
(foo (bar (foo (bar 1))))
```

we can construct a combinator expression Z out of the primitive combinators such that A would evaluate to the same result as

```
(((S foo) bar) 1).
```

The possibility to do so is in no sense means that it is easy to find such a Z. For example if we want to define a procedure

```
(define sin+cos
  (lambda (x) (+ (sin x) (cos x))))
```

with the help of combinators we will end up with something like

```
(define curry-k-sin+cos
  (((((((s (k s)) k) (((s (k s)) k) s)) ((s (k s)) k))
     plus) sin) cos))
```

or if we use B, W, C:

```
(define curry-i-sin+cos
  (((((b (b ((b ((b (b w)) c)) (b b)))) b)
     plus) sin) cos))
```

This example demonstrates that people most likely will not accept a programming language based on combinatory logic; nevertheless, it is an interesting example of how more complex functions can be built up from simpler ones by passing procedural arguments to other procedures. We will rely heavily on this methodology throughout this book. **Problem:** What does a function `f6` do if it is defined as

```
(define f6 ((c i) 3))
```

Problem: Implement I, C, W, B in terms of K and S.

2.3.2 Conditionals

The primitive conditional construct in Scheme is:

(`if` *condition consequent alternative*)

where *condition* is a predicate and *consequent* and *alternative* are forms. The *condition* is evaluated and if it returns a true value (anything other than `#f` or `'()`) then *consequent* is evaluated and its value is returned. Otherwise, *alternative* is evaluated and its value is returned. We can define the `IF-combinator` to be:

```
(define IF-combinator
  (lambda (predicate p q)
    (lambda (x)
      (if (predicate x) (p x) (q x)))))
```

Thus, for example, we can define a function which returns the real part of the square root of a real number as:

```
(define real-sqrt
  (IF-combinator positive? sqrt (lambda (x) 0)))
```

Problem: What does the following procedure do?

```
(define foo (IF-combinator odd? 1+ identity))
```

2.3.3 Primitive Recursion

Recursion is another important procedural schema. A recursive procedure is a procedure which calls itself directly or indirectly. In the latter case, the procedure calls another procedure which calls the original procedure, possibly indirectly. If two or more procedures call upon one another, they are said to be mutually recursive. We will see that virtually all useful procedures on integers can be built up from a very simple class of procedures by using recursion.

One of the simplest examples of a recursive procedure is factorial. N factorial (written $N!$), is defined as

$$N! = \begin{cases} 1 & \text{for } N = 0 \\ N(N-1)! & \text{for } N > 0 \end{cases}$$

This definition leads directly to the following recursive implementation of factorial:

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

While the program above does work, it has several defects. First, we note that it is not quite "first class"; its correctness depends on the global binding of `factorial`. Thus, if we write:

```
(define new-factorial factorial)
(define factorial square)
```

then the reference to `factorial` in the last line of the original definition is to the newly defined procedure (now redefined as squaring its argument) and if we now invoke

```
(new-factorial 5)
```

we will find that it returns 80 instead of 120. In particular, the invocation `(new-factorial 5)` will result in the invocation `(* 5 (factorial 4))`. The reference to `factorial` will now be to the new definition, `square`, and hence, this will result in the invocation `(square 4)`, which returns 16. Finally, `(* 5 16)` will return 80.

What we want is to make a recursive procedural object independent of its global name namely, we want to bind the name factorial to the procedural object in the environment of this procedural object.

There is a special form, `letrec`, which will allow us do just that. The syntax of `letrec` is:

`(letrec ((`var_1 $form_1$`) (`var_2 $form_2$`) ...)` exp_1 exp_2 `...)`

`Letrec` works just like `let` except that all the initializing forms are evaluated in an environment extended to include the var_1, var_2, ..., var_n . It is thus possible to make the var_1, var_2, ..., var_n mutually recursive procedures. We can now overcome the previous difficulty by redefining `factorial` as:

```
(define factorial
  (letrec ((fact
            (lambda (n)
              (if (= n 0)
                1
                (* n (fact (- n 1))))))))
    fact)))
```

Here we use `letrec` to define a procedure, `fact`. `Fact` is known only within the scope of the definition of `factorial`. It is thus called a locally defined procedure. In this case, the reference to `fact` is to the locally defined procedure defined here, not to a globally defined procedure. Now, the self-recursive reference is done through the local binding which cannot be affected by changing the global binding of `factorial`. Note that we

could have called this locally defined procedure `factorial`, and there would have been no conflict with the globally defined procedure of the same name defined by the outer `define`, but this is unnecessarily confusing to the reader.

In general, it is usually only meaningful to use `letrec` to define local, mutually recursive, procedures rather than, say, numerical variables since a numeric variable has to be defined before it is used. Thus, for example:

```
(letrec ((a 3) (b (+ a 2))) ...)
```

is an error since, at the time the initial value of `b` is being evaluated, `a` does not actually have a value which can be passed to `+`. On the other hand, it is possible to define one procedure in terms of another without the latter procedure actually having a specific value at the time the definition of the former procedure is being set up.

By defining `factorial` within the scope of the `letrec` we often also gain another advantage. Many Scheme implementations access a locally defined variable more efficiently than they do globally defined variables. Thus, the second implementation of factorial usually will be faster than the first.

2.3.4 Tail Recursion

The above definitions of `factorial` have another flaw. The first time `factorial` is called, the multiplication operation cannot be performed until both operands are available. This, in turn, does not happen until the result of the second call is returned. This continues to be true until, finally, `n` is 0. Until that time, however, intermediate results and arguments must be saved and kept track of, as must the flow of control among invocations of `factorial`. This takes up both time and space. Thus, the procedure runs more slowly and its domain is limited by the size of the memory allocated for the purpose of keeping track of intermediate results.

Let us closely examine what happens when `factorial` is defined as above and then applied to an argument. First, the `define` itself is evaluated in the global environment. This causes the global environment to be extended with a binding of the variable `factorial` to a procedural object (the body of the `lambda` expression, which is the code defining the procedure) and a pointer to the global environment (the place where `factorial`

was defined). If we now apply `factorial` to the argument `5`, a new environment is created. This new environment has `n` bound to `5` and has the global environment as its parent (see Figure A). The body of the `lambda` expression is evaluated in this new environment. This evaluation results in another invocation of `factorial` which causes another new environment to be created, again with the global environment as its parent and with `n` bound to the value `4` (see Figure A). Further recursive calls to `factorial` continue to create new environments of this type, until finally, when `n` is `0`, factorial returns `0` and the chain of recursive calls returns upward with the appropriate values being computed. Note that during the descent through the recursive calls to `factorial`, as `n` takes the values `5`, `4`, `3`, `2`, and `1`, no computation is done but intermediate values of `n` must be stored for later computation.

All of this can be avoided by changing the definition of factorial to:

```
(define factorial
  (lambda (n)
    (letrec ((factorial-loop
              (lambda (i result)
                (if (> i n)
                    result
                    (factorial-loop (+ i 1) (* result i))))))
      (factorial-loop 1 1))))
```

In this case, the variable result explicitly holds the necessary intermediate result and no auxiliary storage is necessary. A procedure like this is called *tail-recursive*, and it both more useful and more efficient than ordinary recursive procedures.

The essential characteristic of a tail-recursive procedure is that the recursive call is the last thing that the procedure does. It is thus unnecessary to save any local environment in order to complete the computation in the calling procedure.

In order to realize the advantages of tail-recursion, however, it is not enough for a procedure to be tail-recursive. The language it is written in must also be able to take proper advantage of this fact. A language which does this is called a properly tail-recursive language. Scheme is a properly tail- recursive language. *** last environment can become garbage because noone refers to it *** and does not extend the environment when evaluating

the last form in a procedure. Therefore, when the last form in a procedure is a recursive call to the procedure (as will be the case when the procedure is tail- recursive), no new binding is created.

Tail recursion is really iteration. For example, we can write:

```
(define sum
  (lambda (n)
    (letrec ((sum-loop

       (lambda (i result)
                 (if (> i n)
                     result
                     (sum-loop (+ result i) (+ i 1))))))
      (sum-loop 1 0))))
```

we get a procedure which finds the sum of the integers from 1 to n using an iterative loop. This iteration is intrinsically no less (and no more) efficient than using a for-loop in C, or PASCAL or BASIC. This tail-recursive procedure is not *like* an iterative procedure, it *is* an iterative procedure.

2.3.5 Transforming Primitive Recursion to Tail Recursion

We can now ask what are the conditions that allow us to find a tail recursive representation of a primitive recursive procedure. It is possible to prove that any primitive recursive function has a tail recursive form. In Scheme we can construct the best possible proof of all: we can implement a procedure which does the transformation of a primitive recursive procedure into a tail recursive form. We shall restrict ourselves to functions of one variable.

First, we define a procedure, called a *maker* that makes a primitive recursive procedure, given a transformation and an initial value:

```
(define make-primitive-recursive
  (lambda (transform initial-value)
    (letrec
      ((primitive-recursive
         (lambda (n)
           (if (= n 0)
```

```
                initial-value
                (transform n (primitive-recursive (- n 1)))))))
        primitive-recursive)))
```

We can produce an equivalent iterative procedure with:

```
(define make-primitive-iterative
  (lambda (transform initial-value)
    (lambda (n)
      (letrec
        ((loop
           (lambda (i result)
             (if (= i n)
                 result
                 (loop (+ i 1) (transform (+ i 1) result))))))
        (loop 0 initial-value)))))
```

Note, as mentioned above, that the primitive recursive version descends through values of `n` while the iterative version ascends from `n` equal `0` without any requirement for auxiliary storage. In the latter case, the variable result holds the intermediate value of the computation.

Problem: Define `factorial` with the help of `make-primitive-recursive`.

Problem: With the help of `make-primitive-recursive` and `make-primitive-iterat` implement procedures `(make-add-select predicate)` and `(make-add-select-iterativ predicate)` that return a procedure defined on non-negative integers such that for any integer n it returns the sum of those integers less-or-equal to n that satisfy predicate.

Problem: Define `add-odd` as `(make-add-select odd?)` and `(add-odd-iterative)` as `(make-add-select-iterative odd?)`; what is the smallest integer, i, in your system such that `(add-odd-iterative` i`)` runs and `(add-odd` i`)` does not?

We now consider a somewhat more complex form of recursion. Suppose that we have a recursive procedure, p, which requires two previous values returned by itself. The well-known Fibonacci function gives rise to such a procedure. In a manner similar to that described above, we can create a maker which returns a recursive procedure:

```
(define make-two-recursive
```

```
(lambda (transform value-0 value-1)
  (letrec
    ((two-recursive
       (lambda (n)
         (if (= n 0)
             value-0
             (if (= n 1)
                 value-1
                 (transform n
                            (two-recursive (- n 1))
                            (two-recursive (- n 2))))))))
     two-recursive)))
```

Here, two preceding values are passed along as arguments, instead of the one which was passed before. Clearly, this technique can be extended to procedures requiring any specific number of previous values. Again, however, as with composition, the generalization to an arbitrary number of values is difficult.

A corresponding iterative procedure is:

```
(define make-two-iterative
  (lambda (transform value-0 value-1)
    (lambda (n)
      (letrec ((loop
                 (lambda (i first second)
                   (if (= i n)

                       first
                       (loop (+ i 1)

                             (transform (+ i 1) first second)

                             first)))))
          (if (= n 0)
              value-0
              (loop 1 value-1 value-0))))))
```

In addition to the advantages cited above, the iterative version has an enormous advantage over the recursive version in this case. The recursive

version runs for an exponential amount of time as a function of `n`, while the iterative version has runtime linear in `n`. This is due to the fact that the recursive version recomputes prior procedural values many times while the iterative version only computes them once.

From the above discussion, we observe that the transformation from a primitive recursive procedure to a tail recursive procedure is essentially mechanical. Any primitive recursive procedure can be created using an appropriate recursive maker of the above type and a corresponding iterative procedure can be created using a corresponding iterative maker.

Problem: Define a procedure `(fib n)` which returns `n`-th fibonacci number with the help of `two-recursive`.

Time `(fib 20)`.

Problem: Transform `fib` into an iterative function with the help of `two-recursive-iterative`.

Time `(fib 20)`.

2.3.6 Exponentiation—An Example of an Operator

We illustrate the use of operators by considering the problem of creating an exponent in its general mathematical sense. We are given a set, S, and an associative binary operation with identity which is closed over S (monoid). We are also given a non-negative integer exponent, n, and an element, a, in S. The quantity a^n is then defined inductively by:

$$a^n = \begin{cases} 1 & \text{for } n = 0 \\ a * a^{(n-1)} & \text{for } n > 0 \end{cases}$$

We use $*$ in the above to denote an operation which is part of the definition of the exponent and 1 to denote the identity of this operation.

The conventional notion of exponentiation is obtained by using numbers as the set and multiplication as the operation. There are, however, many other meaningful operations and sets for which the definition of exponentiation is interesting. We may wish to do multiplication modulo p. This example is explored below. Finding the n-th power of a matrix is another example. By suitably defining an inner product to be used within the matrix multiplication operation, it is possible to find shortest paths in a network via matrix multiplication.

In all of these examples, the basic notion of exponentiation is the same; we wish to apply operation repeatedly to the same argument. We wish to capture this in a procedure without making the procedure dependent on the physical representation of a or on any properties of operation except those mentioned above.

We begin by defining the procedure `make-exponent` along the lines of the above inductive definition. This procedure takes an `operation` and its `identity` as arguments and returns another procedure as its value. The procedure returned by `make-exponent` is an exponentiation procedure which takes an exponent (the value of n) and value (the value of a) as arguments. The procedure starts with the `identity` in its accumulator and then invokes `loop` repeatedly applying `operation` to `value` and the result of previous operations.

Note that `make-exponent` is not itself an exponentiation procedure, but rather, that it returns an exponentiation procedure. We refer to procedures like this as *makers* and will discuss them in more detail in the next section.

Thus, `make-exponent` is called once to create the exponentiation procedure and the exponentiation procedure is then, in general, called many

times during the course of computation. This is part of the style of higher order programming.

```
(define (make-exponent operation identity)
  (lambda (value exponent)
    (define (loop accumulator i)
      (if (= i 1)
          accumulator
          (loop (operation value accumulator) (- i 1))))
    (if (= exponent 0)
        identity
        (loop value exponent))))
```

Thus,

```
(define f1 (make-exponent * 1))
```

defines ordinary exponentiation,

```
(define f2 (make-exponent + 0))
```

(somewhat obtusely) defines multiplication and

```
(define f3 (make-exponent mat-mult identity-matrix))
```

defines the n-th power of a matrix, where `mat-mult` is a matrix multiplication procedure and `identity-matrix` is an identity matrix (or a procedure which creates an identity matrix) of the right size for the operand; i.e., if a is an n-by-n matrix, then `identity-matrix` must also be n-by-n. Thus, unlike in the other cases, the identity is not strictly a property of operation. Nevertheless, as we will see in the next section, using makers it is possible to define an appropriate exponent in a specific situation.

The procedure returned by `make-exponent` invokes operation $n - 1$ times. It is actually only necessary to do on the order of $\log n$ operations. This is accomplished by repeatedly squaring the intermediate result rather than simply multiplying the result by `value`. We rely on the associativity of `operation` to justify the rearrangement

```
( a * ( a * ( a * a ) ) ) = ( ( a * a ) * ( a * a ) )
```

which is central to the procedure make-binary-exponent given below.

The procedure make-binary-exponent proceeds in two phases. First, `even-loop` is invoked to repeatedly square `value` and halve `exponent` until `exponent` becomes odd. Then, `odd-loop` is invoked to compute this odd power of a. Thus, the original exponent, n, is represented as $2^k q$, where q is an odd integer. `Even-loop` computes `value * 2`^{k} and `odd-loop` then raises this to the k-th power. If p is the number of bits in the binary representation of q (i.e., p is the smallest integer not less than $\log q$ and m is the number of 1's in the binary representation of q, then `odd-loop` invokes `operation` $p+m-1$ times. `Even-loop` invokes `operation` k times. Thus, if we let $p + k = L$, then L is the ceiling of $\log n$, and `operation` is invoked a total of $t = L + m - 1$ times. Clearly, m is no greater than L. We thus have that:

$$L \leq t \leq 2L$$

So `operation` is invoked on the order of $\log n$ times. Since every invocation of `operation` can at most double the power of a, at least L invocations are required. We see, therefore, that `make-binary-exponent` is within a factor of two of the optimum. Actually, Knuth [], in an extended discussion of this shows that this algorithm is very close to the optimum in terms of the number of invocations of `operation` required, rarely missing by more than one or two. Gonnet [] shows that it is always within a factor of ... of the optimum.

```
(define (make-binary-exponent operation identity)
  (lambda (value exponent)
    (define (even-loop value exponent)
      (if (even? exponent)
          (even-loop (operation value value)
                     (quotient exponent 2))
          (odd-loop value value (quotient exponent 2))))
    (define (odd-loop accumulator value exponent)
      (if (= exponent 0)
          accumulator
          (let ((next-value (operation value value)))
            (odd-loop
              (if (odd? exponent)
                  (operation accumulator next-value)
```

```
                accumulator)
            next-value
            (quotient exponent 2)))))
    (if (= exponent 0)
        identity
        (even-loop value exponent))))
```

Problem: What is `(make-binary-exponent + 0)`?

2.3.7 Factorization - Another Example of a Maker

We now consider the problem of defining a procedure, which we will call `factors`, to find prime factors of a given number, n. We will limit the search for factors to numbers no greater than a given limit. `Factors` works by testings trial factors of n starting from 2. We test a trial factor, i, by checking if n modulo i is 0. If i is found to be a factor of n, i is factored out and the procedure continues, trying to find factors of n/i. The built-in procedure `cons`, which adds an element to the head of a list, is used to build (construct) the list of factors. `Cons` is part of extensive list-processing capabilities which are built into Scheme and will be discussed in greater detail in Chapter 3.

The trial factors are generated by a procedure which we refer to as `generator`. We use a simple generation procedure here which generates the next odd number not divisible by 3 by alternately taking steps of 2 and 4. The creation of this procedure is interesting in its own right.

`Generator` has two state variables, `d` and `step`, which retain their values from one call to the next and are not directly accessible outside the procedure. This is an example of an *encapsulation*, where a local environment is maintained within a procedure and state variables maintain the local state which is used on successive calls to generate the desired value. The nature of this local state and the exact mode of computation are totally hidden from procedures which use it. In particular, it is possible to replace `generator` by a more sophisticated procedure which skips numbers which are factors of 5 and 7 as well, or even to replace it by one which skips to the next prime number, without modifying `factors` in any way. We will use encapsulations extensively in the remainder of this text.

A problem arises, however, in the implementation of `generator`. As defined above, it will generate the next number in the sequence given the

current number, d, and the current value of step. Somehow, however, these values must be initialized (in this case, to 1 and 4, respectively.) It is not sufficient to simply define these values in a let within generator, however, because d and step would then be initialized once when the define is evaluated and we would be unable to restart the sequence if we wanted to factor a second number. We would be left with a very subtle bug. The first procedure to use generator would work properly, but only the first time it was called. This is clearly not acceptable.

The solution to this problem is to define a maker; i.e., a procedure which creates the desired procedure and does the necessary initialization. The procedure make-generator given below does just that. It uses let to initialize d and step and then returns the required procedure within the environment in which these initializations took place. Thus, every time make-generator is invoked, a newly initialized instance of the procedure is created and a new sequence is begun.

```
(define (make-generator)
  (let ((d 1)
        (step 4))
    (lambda ()
      (cond ((= d 1) (set! d 2))
            ((= d 2) (set! d 3))
            ((= d 3) (set! d 5))
            (else
              (set! step (if (= step 4) 2 4))
              (set! d (+ d step))))
      d)))
```

Given make-generator, it is now possible to define factors properly. A maker, however, can be used again to good advantage. As defined below, factors is a function of two arguments, the number to be factored and the limit on the size of factors. In different applications, we might choose different generators. We use the maker, make-factors, to define a specific instance of factors, given a specific make-generator.

```
(define (make-factors make-generator)
  (lambda (number limit)
    (define generator (make-generator))
```

```
(define (loop n i result)
  (cond ((= n 1) result)
        ((> i limit) (cons n result))
        ((= (modulo n i) 0)
         (loop (quotient n i) i (cons i result)))
        (else (loop n (generator) result))))
(loop number (generator) '()))
```

For example, if we wanted to define a procedure to find all factors no greater than 7 using the generator defined above, we would write:

```
(define factors (make-factors make-generator))
```

If we then invoked

```
(factors 18018 7)
```

it would return

```
(143 7 3 3 2)
```

Note that the factors come out largest first because cons adds elements to the front of the list and that the unfactored portion of the number, returned at the head of the list, includes all the factors greater than 7, in this case 11 and 13.

2.3.8 Primality Testing

We now turn to the problem of testing if a number is prime. Using the generator defined above, a simple primality tester can be constructed. We use the generator to select potential factors of p and if none are found, then p is prime. We need only test for factors up to the square root of p.

```
(define (prime? p)
  (define generator (make-generator))
  (define (loop i limit)
    (cond ((= (modulo p i) 0) #!false)
          ((>= i limit) #!true)
          (else (loop (generator) limit))))
  (loop (generator) (integer-sqrt p)))
```

It is interesting to note in passing that for sufficiently large p, the built-in square root procedure is inadequate for finding the place to stop. It returns a floating point number which may not have sufficient precision to accurately define the square root. With versions of Scheme which implement exact arithmetic for integers, it is possible to obtain the square root of a number exactly using only integer operations:

```
(define (integer-sqrt number)
  (define (loop guess)
    (let ((new-guess (quotient number guess)))
      (if (<= (abs (- new-guess guess)) 1)
          (min guess new-guess)
          (loop (quotient (+ guess new-guess) 2)))))
  (if (<= number 1)
      number
      (loop (floor (sqrt number)))))
```

Some applications, most notably those in the area of encryption, require that we find prime numbers with hundreds of digits. It is clear that any method which relies on testing primality explicitly by division is doomed to failure in this case. Fortunately, there is an alternative which requires a much smaller number of tests, actually, a constant number independent of p. This approach does not guarantee that p is prime, but instead provides us with a bound which states that p is prime with probability which can be made as close as we want to 1.

The method is based on several observations from number theory which we now state. In the discussion in the remainder of this section, all arithmetic is modulo p. The first fact, called the Little Fermat Theorem, is that

$$a^{p-1} = 1$$

for all a such that $0 < a < p$ if p is prime. The converse is also true, as we shall see later. To see that this is true, consider the numbers

$$1, 2, \ldots p-1$$

Consider any a such that $0 < a < p$. We multiply each of these numbers by a, obtaining

$$a, 2a, 3a, \ldots, (p-1)a$$

These numbers must all be different modulo p for if not, say if ia and ja were the same, then

$$a(i-j) = 0$$

and a would be a factor of p, contradicting the assumption that p is prime. Also, none of the ia can be 0 since, again, if this were so then a and i would be factors of p. Therefore, modulo p all the elements of the second sequence are different and are non-zero. Hence, the second sequence is just a permutation of the first. We thus have, modulo p:

$$\prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} ia = a^{p-1} \prod_{i=1}^{p-1} i$$

So,

$$a^{p-1} = 1$$

We now show that the converse is also true. If p is not prime then there exist f and g both larger than 1 and less than p such that f and g are factors of p and hence

$$fg = 0$$

If we assume that $a^{p-1} = 1$ for all positive a less that p (in particular, for $a = f$ and $a = g$) then, if we now consider $(fg)^{p-1}$ we have

$$0 = (fg)^{p-1} = f^{p-1} * g^{p-1} = 1 * 1 = 1$$

an obvious contradiction. Hence, we have that

$$a^{p-1} = 1$$

for all positive a less that p if and only if p is prime.

Thus, it is possible to test for primality of a given number p by raising all numbers $a < p$ to the $(p-1)$-th power. It is in fact sufficient to test only prime numbers less than p since the prime factors of any composite number which fails the test will also fail the test. This still results in and unacceptably large number of tests, however.

It is in fact unlikely, based on empirical evidence, that a random a will pass the test if p is not prime, but unfortunately there is no provable bound on how unlikely this is.

The procedure `fermat?`, defined below, tests a number, p, for primality using a single a as defined above.

```
(define (fermat? p a)
  (define (times x y) (modulo (* x y) p))
  (define exponent (make-binary-exponent times 1))
  (= (exponent a (- p 1)) 1))
```

By calling fermat? a sufficient number of times (still a small constant independent of p) it is possible to test for primality with virtual certainty. We thus have a primality test which has a running time which is polynomial in the length of the number being tested.

The Fermat test can be strengthened so that it is now possible to prove an upper bound on the probability of its being fooled. The improved test relies upon the following simple number theoretic fact. For any positive a and p:

> If $a^2 = 1 \pmod{p}$, then either $a = 1 \pmod{p}$
> or $a = p - 1 \pmod{p}$
> or p is not prime.

Suppose $a^2 = 1 \pmod{p}$. We then have:

$$(a+1)(a-1) = 0$$

This can be true only if at least one of $a+1$ and $a-1$ is 0 modulo p or if $a+1$ and $a-1$ are factors of p.

This leads to the following procedure, `rabin?` [Ref?], which tests p, an odd number greater than 2, for primality using a single number, a. As before, if a passes the test, we are probabilistically assured that p is prime. If the number fails, however, we are certain it is not prime. `Rabin?` proceeds along the same general lines as `fermat?`, but it was shown by Rabin and Weinberger [ref] that for a non-prime p and a random a, the probability of error is no greater than 1/4. Because of this, if we repeat the test T times, the probability of error is bounded by $1/2^{2T}$. Empirically, it has been found that the probability of error is far smaller.

Since p is odd and greater than 2, we can write `p-1` as:

$$p - 1 = n = 2^k q$$

where q is odd and k is greater than 0. We thus begin to compute a^n. We first compute a^q. If this is 1, then p is prime since

$$a^n = (a^q)^{2^k} = 1$$

In this case, the procedure halts, declaring p to be prime. If $a^q = p - 1$, then the procedure also halts, declaring p to be prime.

Otherwise, the procedure continues, squaring a^q. If this is 1, then p is not prime since $(a^q)^2$ is 1 but a^q is not 1 or $p - 1$, and, as explained in the observation above, either $a^q - 1$ or $a^q + 1$ is a factor of p. If a^q is $p - 1$, then as we just explained, p is declared to be prime. The procedure continues, replacing a^q by $(a^q)^2$ until either $a^q = 1$ or $a^q = p - 1$ or a^n is computed.

The procedure, `rabin?`, below assumes that the appropriate values of `k` and `q`, as defined above, are already available. A procedure for computing `k` and `q` is given as part of the overall primality testing procedure given after `rabin?`.

```
(define (rabin? p k q a)
  (define (times x y) (modulo (* x y) p))
  (define exponent (make-binary-exponent times 1))
  (define (loop k z j)
    (if (= z (- p 1))
        #!true
        (if (or (>= j k)
                (= y 1))
            #!false
            (loop k (times z z) (+ j 1)))))
  (let ((z (exponent a q)))
    (if (= z 1)
        #!true
        (loop k z 1))))
```

It is now possible to define a very fast test for primality, which can be made as accurate as we desire. We begin by defining a maker which creates a primality tester given the desired number of tests. The procedure created generates the appropriate number of random test numbers and calls `rabin?`. The local procedure `outer-loop` finds the appropriate values of `k` and `q` as defined above; i.e., `k` and `q` such that

$$n = p - 1 = 2^k q \quad \text{for } q \text{ odd}$$

Note that `outer-loop` produces values which are functions only of `p`, not the random test numbers, and hence needs to be executed only once.

```
(define (make-fast-prime number-of-tests)
  (lambda (p)
    (define (outer-loop k q)
      (if (even? q)
          (outer-loop (+ k 1) (quotient q 2))
          (inner-loop 1 k q)))
    (define (inner-loop i k q)
      (if (> i number-of-tests)
          #!true
          (if (rabin? p k q (+ (random (- p 2)) 2))
              (inner-loop (+ i 1) k q)
              #!false)))
    (outer-loop 0 (- p 1))))
```

We can then define the primality tester:

```
(define fast-prime? (make-fast-prime 25))
```

This functions so efficiently that it is even possible to embed it in a loop to find prime numbers by testing successive odd numbers starting at a given point until a prime is found. It can be shown [Ref] that on the average $(\ln n)/2$ numbers will be tested before a prime number is found. Thus, it is reasonable to use the following procedure to find even very large primes; e.g., for n on the order of 10^{100}.

```
(define (make-first-prime-larger number-of-tests)
  (lambda (n)
    (define fast-prime? (make-fast-prime number-of-tests))
    (define (loop i)
      (if (fast-prime? i)
          i
          (loop (+ i 2))))
    (if (odd? n)
        (loop n)
        (loop (+ n 1)))))
```

Problem: In 1644, the French mathematician Marin Mersenne conjectured that numbers of the form $2^p - 1$ were prime for $p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$, and for no other $p < 257$. It is now easy to test his conjecture. Write a progam to test whether he was correct.