4.5 THE CLASS OF PRIMITIVE RECURSIVE FUNCTIONS

Primitive recursion is a powerful operation that can be used to obtain a surprisingly extensive class of functions when used with composition and the basic functions of Section 4.3. We illustrate its application in the following list of some of the functions obtained through the use of primitive recursion and composition (Davis [3]).

1. $\Sigma(x, y) = x + y$

$$\begin{cases} \Sigma(0, y) = y \\ \Sigma(x + 1, y) = S(\Sigma(x, y)) \end{cases}$$

2. $\Pi(x, y) = x \cdot y$

$$\begin{cases} \Pi(0, y) = 0 \\ \Pi(x + 1, y) = \Pi(x, y) + y = \Sigma(\Pi(x, y), U_2^{(2)}(x, y)) \end{cases}$$

3. $P(x) = (x - 1) \quad \text{for } x \geq 1$
$\quad\quad = 0 \quad\quad\quad\ \text{for } x = 0$

$$\begin{cases} P(0) = 0 \\ P(x + 1) = x = U_1^{(1)}(x) \end{cases}$$

4. $$D(x, y) = \begin{cases} x - y & \text{for } x \geq y \\ 0 & \text{for } x < y \end{cases}$$

This last function is the same as the ordinary difference $x - y$ when the result is a natural number. However, negative numbers are excluded from its

range and it is defined as 0 when the ordinary difference is negative. It is obtained as follows by primitive recursion and composition from the preceding functions in our list.

$$\begin{cases} D(x,0) = x = U_1^{(1)}(x) \\ D(x,y+1) = P(D(x,y)) \end{cases}$$

$D(x,y)$ is often written as $x \dot{-} y$.

5. $E(x,y) = x^y$

$$\begin{cases} E(x,0) = 1 \\ E(x,y+1) = E(x,y) \cdot x = E(x,y) \cdot U_1^{(2)}(x,y) = \Pi(E(x,y), U_1^{(2)}(x,y)) \end{cases}$$

6. $F(x) = x!$

$$\begin{cases} F(0) = 1 \\ F(x+1) = (x+1) \cdot F(x) = S(x) \cdot F(x) = \Pi(S(x), F(x)) \end{cases}$$

7. $A(x,y) = |x - y| = D(x,y) + D(y,x)$
$= \Sigma(D(x,y), D(U_2^{(2)}(x,y), U_1^{(2)}(x,y)))$

The list can be extended considerably, and the class of functions so generated is called the *primitive recursive functions.*

DEFINITION: *The class of primitive recursive functions consists of those functions that can be obtained by repeated applications of composition and primitive recursion starting with (1) the successor function, $S(x) = x + 1$, (2) the zero function, $N(x) = 0$, (3) the generalized identity functions, $U_i^{(n)}(x_1, \ldots, x_n) = x_i$.*

All of these functions are easily shown to be total since the three basic functions are total and every single application of either composition or primitive recursion applied to total functions can only yield a total function.

First used by Gödel in 1931, this class of functions was called by him the recursive functions. This latter term now has a different, more general meaning, which will be described in the sequel.

The question arises as to whether the class of primitive recursive functions can be equated with the effectively computable (total) functions. It can not be so identified, but it is far from easy to construct a computable function that is not primitive recursive. Almost all of the functions computed on computers (from an earlier remark this can be construed to refer as well to all data-processing operations) are primitive recursive. However, we shall see below a rather bizarre example, Ackermann's function, of a function that is effectively computable but is not primitive recursive. The computation of this function has often been done on machines as a programming tour de

force, but it is an interesting question as to whether Ackermann's function is the *only* effectively computable, but not primitive recursive, function that has ever been programmed on digital computers.

4.6 ACKERMANN'S FUNCTION

As extensive as is the class of primitive recursive functions, it does not include the function defined by the following equations:

$$\begin{cases} f(0, n) = n + 1 & (1) \\ f(m, 0) = f(m - 1, 1) & (2) \\ f(m, n) = f(m - 1, f(m, n - 1)) & (3) \end{cases}$$

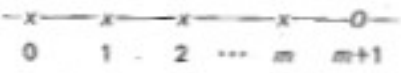
This is an example of a "doubly-recursive" function and is a special case of a class of functions defined by W. Ackermann [1] and shown by him not to be primitive recursive.

It may not be easy at first glance, especially for the person without much programming experience, to see that equations (1), (2), and (3) provide a proper definition of a computable function $f(m, n)$, a definition that enables one to write a program to compute $f(m, n)$. The reader should compute $f(2, 2)$ if he has not already done this exercise from Chapter 2. The computation will take only a few minutes.

The novice programmer will appreciate the built-in intelligence of a compiler that can cope with this function definition. For example, a PL/I procedure to compute $f(m, n)$ is given below.

```
F: PROCEDURE (M,N) RECURSIVE RETURNS (FIXED);
   DECLARE (M,N) FIXED;
   IF M = 0 THEN RETURN (N + 1);
   ELSE IF N = 0 THEN RETURN (F(M - 1, 1));
   ELSE RETURN (F(M - 1, F(M, N - 1)));
   END F;
```

We shall see that the given equations (1), (2), (3) define a procedure that generalizes that of primitive recursion. The operation of primitive recursion for the definition of a function can be described with reference to the following diagram.



The operation is applied iteratively on one argument. Given the value of the function at $x = 0$, we can step along the number line computing the

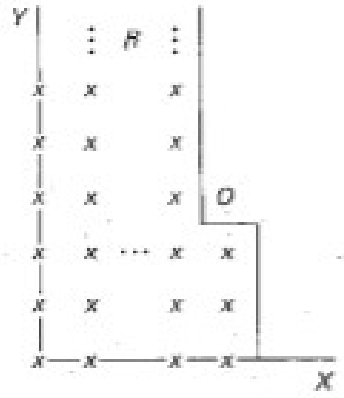


Fig. 4.3 The "doubly recursive" pattern for the computation of Ackermann's function.

value of the function at the next point $x = m + 1$ in terms of the preceding functional value at $x = m$.

Equations (1), (2), (3) enable us to do something similar but in *two* dimensions. With reference to Fig. 4.3, note that we can find, from equation (1), the value of $f(x, y)$ at any point $(0, n)$. Equation (2) gives the value of $f(x, y)$ at $(m, 0)$ if we have previously found the value of $f(x, y)$ at $(m - 1, 1)$. Equation (3) determines $f(x, y)$ at $(m + 1, n)$ if we know the value of $f(x, y)$ at $(m + 1, n - 1)$ and can find $f(x, y)$ at any point on the line $x = m$.

Taking all these details into account we are able to compute the value of $f(x, y)$ at the point labeled O in Fig. 4.3, if we are able to compute the function at any point marked x in region R. From all these remarks, it follows that we can find $f(x, y)$ at every lattice point in the first quadrant of the (X, Y) plane. The description of this *two-dimensional* schema as being *doubly recursive* is justified.

A surprisingly simple flowchart that realizes the procedure we have described is shown in Fig. 4.4. In the course of applying the algorithm to find $f(x, y)$ at the point 0 of Fig. 4.3, we need to obtain the values of $f(x, y)$ at other points in the region R of Fig. 4.3. For example, the first reduction of $f(2, 2)$ involves equation (3), and we need to obtain $f(2, 1)$. Those pairs of coordinates at which the function is needed as the algorithm is elaborated are stored, as they occur in the computation, in a *pushdown store*—a list where the last element recorded is the first one to be read out by the program when the list is accessed (a "LIFO" or "last in first out" list). We call this store the "needed function values stack" (NFVS).

When the NFVS is read, the word at the top will be read and will disappear from the stack, the one below it moving to the top. When a word is

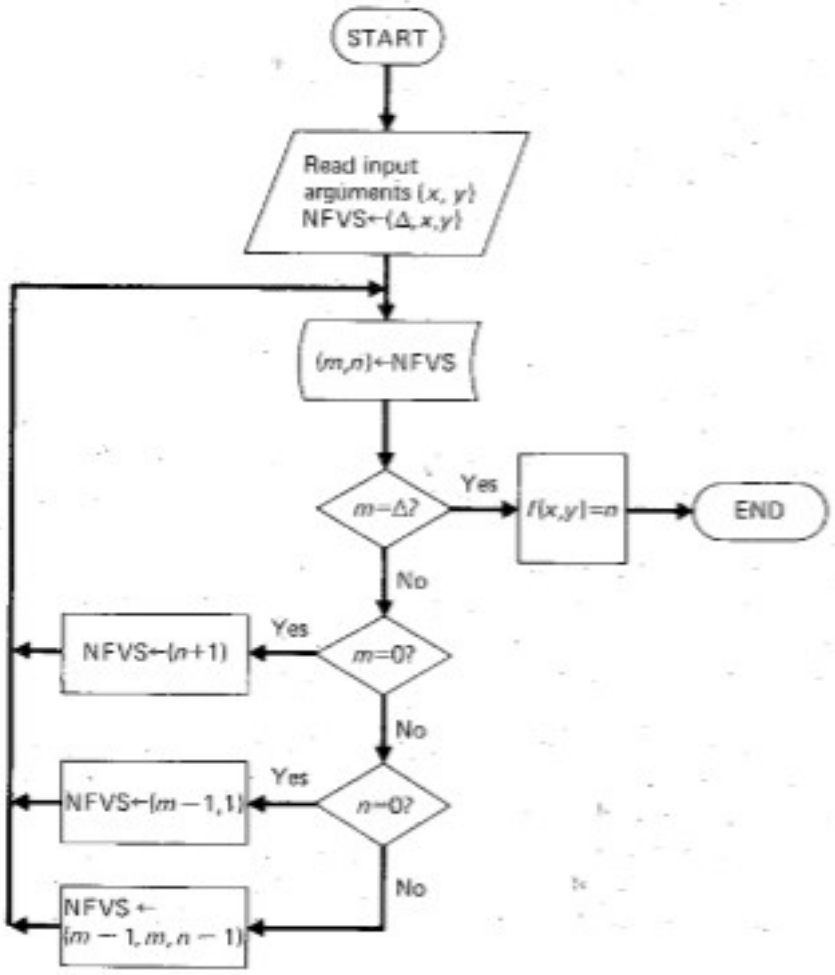


Fig. 4.4 A flowchart for the computation of Ackermann's function.

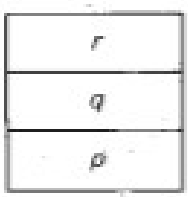
read we say that we "pop" the stack. The language is, of course, used by analogy to a real stack, such as of cafeteria trays. (See the discussions of "pushdown automata" and of "stack automata" in Chapter 7.)

The pushdown store finds wide application in compiling. For example, it is easily seen that such a device is advantageous in interpreting a string written in the postfix notation defined in Chapter 2. More generally, we shall see examples in Chapter 10 of the use of pushdown stacks in the compiling process.

When we write

$$\text{NFVS} \leftarrow (p, q, r),$$

this will mean that the words p, q, r are written, in that order, on the stack. If we start with an empty stack, then, after the execution of this operation, the NFVS will look like



The first word read on a subsequent READ operation will be r. If we execute

$$(a, b, c) \leftarrow \text{NFVS},$$

this shall be taken to mean that r, the first word read, will be stored in c, q in b, p in a, with a similar interpretation for a list of any length.

The procedure described in Fig. 4.4 makes use of a special symbol, or word, Δ that marks the bottom of the stack and is the first word written on an empty stack.

We leave it to the reader to recognize that this procedure realizes the computation of the defining equations for Ackermann's function.

The procedure can be easily programmed on any general purpose computer, given enough storage. (It will be easier, however, if we assume some bound, like one machine word, on the size of the arguments and on all intermediate computed quantities.) It thus appears that this function is effectively computable, and we can compute its value for any given pair of arguments.

However, it is a most remarkable function, and perhaps we should examine more closely our use of the phrase "we can compute." To find, say, $f(2, 2)$ is a trivial computation and $f(3, 3)$, $(= 61)$, can be computed in a tiny fraction of a second on any modern computer.

But $f(4, 4)$ is another story. No one knows with certainty at this time if the universe is finite or infinite. However, current theories with some reasonable evidence to support them (Relativity and the "Big Bang" theories) seem to imply that the universe is finite. It is enormous, however, and we are fairly certain of the existence of many billions of galaxies, most of them containing many billions of stars. The total mass of this finite (if it is) cosmos has been estimated to include something like 10^{80} elemental particles.

If every one of these particles were used in some way to represent one digit in the decimal representation of $f(4, 4)$, not only would they not be sufficient, but *we would not even be able to represent that number which is the*

number of digits in f(4, 4) (i.e., $[\log_{10} f(4,4)]$). What meaning then is there the statement that "we can compute" this function?

It is not very difficult to show that $f(4,4)$ is the same as

$$2^{2^{2^{2^{2^{2}}}}} - 3$$

This is an enormous number, larger than the decimal number 1 followed $10^{19,199}$ zeros. It may seem strange, looking at Fig. 4.4, that $f(x,y)$ can be so large for such small arguments. For, only one box involves any increasing arithmetic operation and that is simply the addition of 1.

We shall omit the proof here that this function, although computable, not primitive recursive. The argument depends upon the fact that $f(x,x)$ grows more rapidly than any given primitive recursive function. The reader will have an opportunity to prove it as one of the exercises.

4.7 THE CLASS OF PARTIAL RECURSIVE FUNCTIONS

The class of primitive recursive functions can be extended by introducing a additional generating rule.

Minimalization. The operation of minimalization introduces for each given total function $f(y, x_1, x_2, \ldots, x_n)$ a new function $h(x_1, x_2, \ldots, x_n)$ whose value for given $(x_1, x_2, \ldots, x_n)$ is the least y such that $f(y, x_1, x_2, \ldots, x_n) = 0$. We write $h(x_1, x_2, \ldots, x_n) = \min\, y[f(y, x_1, x_2, \ldots, x_n) = 0]$. Fig. 4.5 shows the schema for a function f that involves a single x.

Example. Let $z = [x/2]$ denote the largest integer that does not exceed $x/2$ (sometimes called the "floor" of $x/2$ and written as $\lfloor x/2 \rfloor$; the "ceiling" of $x/2$, written as $\lceil x/2 \rceil$, is the smallest integer not exceeded by $x/2$). We shall define $z = [x/2]$ by minimalization.

We have

$$z = [x/2] = \min y(2(y+1) > x).$$

This is not yet recognizable as an application of minimalization since the expression in parentheses is not of the form $f(y, x) = 0$. We continue:

$$\begin{aligned} &= \min y(2y + 2 \geq x + 1) \\ &= \min y(2y + 1 \geq x) \\ &= \min y(x \dot{-} 2y \dot{-} 1 = 0). \end{aligned}$$

This last expression is clearly recognizable as an application of minimalization (we have used the fact that $a \geq b$ is equivalent to $b \dot{-} a = 0$).

The function $h(x_1, x_2, \ldots, x_n)$ is not necessarily a total function since there may not be any value of y for which $f(y, x_1, x_2, \ldots, x_n)$ is equal to zero. A programmed search for such a y will never terminate.

Thus, if

$$h(x) = \min y(\,|\,y^2 - x\,| = 0),$$

$h(x)$ is defined only for arguments x which are perfect squares.

The operation of minimalization, then, may lead to a *partial*, and not total, function when it is applied to a total function.

Comparing Fig. 4.5 with Fig. 4.2, we note that in the case of primitive recursion we are assured of exiting from the loop after some prescribed number of times, known when we enter the loop. However, in the operation of minimalization there is no way of knowing in general at its entry point whether the loop will terminate or how many times it will be executed.

If we extend the class of primitive recursive functions by allowing the operation of minimalization (clearly a computable, or programmable, operation), we generate a larger class of functions called the *partial recursive functions*. This class includes functions that are not primitive recursive, not only because the new functions obtained may not be total, but it also includes total functions that are not primitive recursive.

Now that, through minimalization, we have introduced functions that are not necessarily defined for all arguments, we must reconsider the meaning of the operation of composition, because the functions $f(y_1, \ldots, y_m)$, $g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)$ to which it is applied may not be total. We shall say that the function $h(x_1, \ldots, x_n)$, obtained by replacing each y_i in f by $g_i(x_1, \ldots, x_n)$, is defined at a point $x = (x_1, \ldots, x_n)$ if and only if (1) all the g_i are defined at x and (2) f is defined at $(y_1, \ldots, y_m)$ where $y_i = g_i(x_1, x_2, \ldots, x_n)$.

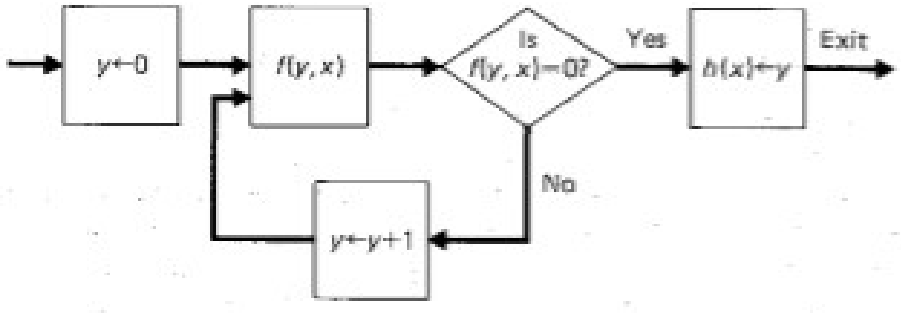


Fig. 4.5 The operation of minimalization—computation of $h(x)$ from $f(y, x)$.

DEFINITION: *The class of partial recursive functions consists of those functions that can be obtained by repeated applications of composition, primitive recursion, and minimalization, starting with* (1) *the successor function,* $S(x) = x + 1$, (2) *the zero function,* $N(x) = 0$, (3) *the generalized identity functions,* $U_i^{(n)}(x_1, \ldots, x_n) = x_i$.

It can be shown (Davis [3]) that the operation of primitive recursion can be dispensed with in defining the class of partial recursive functions *if* we extend the list of assumed basic functions to include (4) $\Sigma(x, y) = x + y$, (5) $D(x, y) = x \dot{-} y$, (6) $\Pi(x, y) = x \cdot y$. That is, the class of partial recursive functions consists of those functions that can be obtained by repeated applications of composition and minimalization starting with these six given basic functions.

The *recursive* functions (sometimes called the *general recursive* functions) are those total functions which are included in the class of partial recursive functions. Ackermann's function is an example of a recursive function that is not primitive recursive. In programming computers we must try to avoid entering arguments that are outside the domain of a programmed partial recursive function; we do not want infinite loops or endless procedures.

The class of partial recursive functions is very broad and, it is believed, can be identified with the heuristically and vaguely defined class (cf. Chapter 1) of the partial effectively computable functions. Since there is no precise definition of effective computability, the equivalence of the notions of *(partial) recursiveness* and *(partial) effective computability* cannot be proven. It is, rather, an act of faith substantiated by the fact that no one has been able to describe a function that is generally admitted to be effectively computable but which is not recursive. The formal assertion of this equivalence, when stated for total functions, is known as

> *Church's Thesis:* The class of effectively computable functions can be identified with the class of recursive functions.

That any recursive function is effectively computable can be proven by showing that any function obtained by composition, primitive recursion, and minimalization starting with the basic functions is *programmable*. We leave to the reader the problem of recognizing, from the flowcharts for these operations, that if programs are available to compute functions f, g, etc., then we can construct a program to compute any function formed from them by composition, primitive recursion, or minimalization. The programming is trivial *if* we introduce the simplifying assumption that all arguments and intermediate computed quantities are limited to one machine word (or at least some fixed number of words).

If we accept the equivalence of the notion of effective computability and of recursiveness, then in order to prove that a particular function is recursive it will suffice to show that there is some effective procedure—or some program—that will compute it.

4.8 REMARKS ON THE IMPLICATIONS OF THESE FUNDAMENTAL OPERATIONS TO THE FACILITIES THAT MUST BE PROVIDED IN ANY GENERAL PURPOSE COMPUTING DEVICE OR LANGUAGE

We have seen that, accepting the result that the class of recursive functions can be equated with the class of effectively computable functions, the operations of composition, of primitive recursion, and of minimalization play key roles, and any device that purports to provide a general computing capability must be capable of implementing these operations. The operation of *composition* can be interpreted as the providing of a *subroutine* capability—the ability in any computation to replace an argument by the result of another computation.

The operation of minimalization (or that of iteration, cf. below) involves essentially the ability to program a *loop* with a test to determine exit conditions. It uses the DO WHILE operation of structured progamming, which we describe in a later section. The primitive recursion operation is, as has been noted, not quite as powerful as that of minimalization. It also involves the execution of a loop, but in a situation where the number of passes through the loop *have been predetermined*, as opposed to the use of an arbitrary test at each stage to determine if a terminating condition has been attained.

4.9 A PROGRAMMING LANGUAGE PRECL BASED UPON THE GENERATION OF THE PARTIAL RECURSIVE FUNCTIONS

We have mentioned the use of certain extended programming language features to facilitate the description of the generation of functions by the operations of composition and of primitive recursion. We shall elaborate upon these somewhat and shall describe a simple language with facilities for the definition of function procedures, but only procedures that are restricted to the use of composition, of primitive recursion, or of minimalization. These facilities enable the definition of *any* partial recursive function, and Church's thesis implies that this means the definition of any effectively computable function.

We define a rudimental "language" PRECL (Partial RECursive Language) for the description of the partial recursive functions. This language will provide little more than a formalized upper-case restatement of the

definitions of functions by composition, primitive recursion, and minimalization. The correspondence between these formal statements and what we have written earlier will be obvious.

Assume that there are three system built-in functions. These are:

$$N(X) = 0;$$
$$S(X) = X + 1;$$
$$U\_I\_N\_(X1, X2, \ldots, XN) = XI; (I, N = 1, 2, 3, \ldots, I \leq N).$$

The arguments in these and all other PRECL defined functions are assumed to be natural numbers of any length when these functions are called in assignment statements such as Z = F(X);

The third of the built-in functions, $(U_i^{(n)}(x_1, x_2, \ldots, x_n))$, represents an infinite number of functions corresponding to the possible numeric values assigned to I, N.

PRECL contains procedure definition facilities to allow the definition of functions by composition, primitive recursion, and minimalization. To define a function by composition we write:

```
DEF H(X1, X2, ..., XN) = COMPOS(F(Y1, Y2, ..., YM),
    G1(X1, X2, ..., XN), G2(X1, X2, ..., XN), ..., GM(X1, X2, ..., XN));
```

This is the PRECL equivalent of

$$h(x_1, x_2, \ldots, x_n) = f(g_1(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n)).$$

There are similar PRECL definitions for functions by using primitive recursion or by using minimalization. To define a function by primitive recursion on the first argument we write:

```
DEF R(X1, X2, ..., XN)
                = PRIMREC(F(X2, X3, ..., XN), G(X1, Y, X2, ..., XN));
```

Relating this statement to our earlier notation, this statement defines the function $r(x_1, x_2, \ldots, x_n)$ in accordance with the equations

$$\begin{cases} r(0, x_2, \ldots, x_n) = f(x_2, \ldots, x_n) \\ r(x_1 + 1, x_2, \ldots, x_n) = g(x_1, r(x_1, \ldots, x_n), x_2, \ldots, x_n). \end{cases}$$

Similarly, we write

```
DEF H(X1, X2, ..., XN) = MINIM(F(Y, X1, ..., XN));
```

to denote $h(x_1, x_2, \ldots, x_n) = \min_y(f(y, x_1, \ldots, x_n) = 0)$.

We shall also assume the implicit use of composition, as in most programming languages, in assignment statements where any variable can be replaced by a function defined earlier in a program. That is, we assume we

can write such statements as Z = FUNCT(F1(X, Y), F2(X, Y, F3(X, Y))); with the understanding that when such a statement is executed X, Y will have been assigned numeric values and F1, F2, F3 will all have been defined earlier.

Any function can be defined by a PRECL program that shows its derivation as a sequence of function procedure definitions. After definition, the function can appear in an assignment statement. We can illustrate this by writing the complete program for the computation of the function $A(x, w) = |x - w|$. (Here D(X1, X2) denotes the function $x_2 \dot{-} x_1$.)

```
/ * PROGRAM TO COMPUTE A(X, W) = | X − W | * /
DEF P(X) = PRIMREC(0, U_1_1(X));
DEF D(X1, X2) = PRIMREC(U_1_1(X2), P(Y));
DEF SIGMA(X1, X2) = PRIMREC(U_1_1_(X2), S(Y));
DEF A(X, W) = SIGMA(D(X, W), D(W, X));
GET X, W; PRINT A(X, W);
END;
```

We are not advocating this language for serious consideration as a competitor for COBOL or PL/I but wish merely to emphasize the constructive aspects of the definition of the partial recursive functions.

4.10 REMARKS ON VOCABULARY—"RECURSIVE," "ITERATIVE"

We have noted that the word "recursive" is used with several different, but related, meanings. Because of the importance of these concepts and the frequent usage of the word, we summarize these different interpretations, giving them in order of increasing precision of meaning.

1. A "recursive" definition of anything is a definition of that thing given in terms of itself. We have noted the use of the BNF notation in the recursive definition of a class of symbolic expressions. In this usage new elements of the class being generated are defined in terms of elements known to be in the class. We have also seen the definition of a recursive procedure as a procedure that can call itself.

We have given an example of a recursive subroutine in Chapter 2. As an additional example of a recursive procedure we indicate the following recursive definition of a function in LISP (LISt Processing language). This language includes a wealth of facilities to manipulate lists of expressions, where these expressions may themselves be such lists. It provides very flexible means for recursive definitions.

The example defines a function LENGTH on the domain of lists. If L is a list, LENGTH(L) will denote the number of expressions in L. Thus, if L is

the list $(A\ CB\ (B\ D\ C)\ D)$, the successive expressions of L are A, CB, $(B\ D\ C)$, D, and LENGTH$(L) = 4$. The lambda notation of Chapter 2 is used in LISP to identify functions. If L is a list, the built-in LISP function CDR(L), which maps lists into lists, denotes the sublist obtained by deleting the first element or expression of L. Thus, if $L = (A\ B\ C\ D)$,

$$\text{CDR}(L) = \text{CDR}((A\ B\ C\ D)) = (B\ C\ D).$$

The LISP arithmetic function ADD1 (X), defined for numeric arguments X, gives as result the value of X increased by 1.

```
DEFINE (( (LENGTH (LAMBDA (L)
                  (COND ((NULL L) 0) (T (ADD1 (LENGTH (CDR L))))))) ))
```

The function LENGTH (L) is here defined for a given argument L by scanning the pairs (__) following COND (condition) from left to right. The first element of each pair is a condition, the second element defines the value of LENGTH (L) if that condition is satisfied. Thus, looking at the first pair ((NULL L) 0), if the condition (NULL L) is true (i.e., if the list L is null or empty), then the value of LENGTH (L) is taken to be 0. In the second pair (T (ADD1 (LENGTH (CDR L)))) the condition given is the identically true condition T. This means that the function LENGTH (L), if L is not null, is defined to be LENGTH (CDR L) increased by 1, i.e., one more than the length of the list after deleting the first element of the list. Thus, LENGTH is defined in terms of itself. The definition is recursive. (We have

$$\begin{aligned}\text{LENGTH}\ (A\ B\ C\ D) &= 1 + \text{LENGTH}\ (B\ C\ D) = 2 + \text{LENGTH}\ (C\ D)\\ &= 3 + \text{LENGTH}\ (D) = 4 + \text{LENGTH}\ (\text{NIL}) = 4.\end{aligned}$$

"NIL" denotes the null string.)

2. A function definition is often said to be "recursive" with the specific meaning of its being defined by a direct application of primitive recursion. More generally, the word may apply to any definition of a function where the functional value at an argument x can be found from known functional values at (usually) smaller arguments.

Example. The "Fibonacci numbers" are defined by the equations:

$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_{n+1} = F_n + F_{n-1} \end{cases}$$

These equations define the sequence 0, 1, 1, 2, 3, 5, 8, 13, ...

3. A function is recursive if it is a total function in the class of partial recursive functions. From Church's thesis, this is equivalent to saying that it is computable.

Iteration. An "iterative" procedure is one that is repeatedly applied, and iteration is very close in meaning to that of primitive recursion.

For example, if we are given a number N, it can be shown that the following iterative procedure will determine $\sqrt{N}$ to any desired accuracy.

Choose $x_0 (> 0)$ as a first approximation to $\sqrt{N}$, then compute successively:

$$x_1 = \frac{1}{2}\left(x_0 + \frac{N}{x_0}\right)$$
$$x_2 = \frac{1}{2}\left(x_1 + \frac{N}{x_1}\right)$$
$$\vdots$$
$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{N}{x_i}\right)$$
$$\vdots$$

It is not difficult to show that the successive approximations, or "iterates," x_i approach $\sqrt{N}$ ever more closely, and in fact that the number of correct decimal places in x_i approximately doubles at each step.

For example, if $N = 2$ and $x_0 = 1$, we have:

$$x_1 = \frac{1}{2}\left(1 + \frac{2}{1}\right) = 1.5$$
$$x_2 = \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.416666667$$
$$x_3 = \frac{1}{2}\left(x_2 + \frac{2}{x_2}\right) = 1.414215686$$
$$x_4 = \frac{1}{2}\left(x_3 + \frac{2}{x_3}\right) = 1.414213562,$$

which is correct to 9 decimal places.

At each stage of this procedure we compute the function

$$f(x) = \frac{1}{2}(x + \frac{N}{x})$$

where $x_{i+1} = f(x_i)$.

We can designate the successive iterates, starting with x_0, as:

$$f(x_0), f(f(x_0)), f(f(f(x_0))), \ldots$$

These expressions are often written as:

$$f(x_0), f^2(x_0), f^3(x_0), \ldots$$

To avoid possible confusion with the powers of $f(x)$, we usually write the latter as $(f(x))^2$, $(f(x))^3, \ldots$

The definition of the operation of primitive recursion, applied to a function of a single argument x, includes the equation $r(x + 1) = g(x, r(x))$. If no x appears explicitly in $g(x, y)$ (i.e., outside of $r(x)$, as, for example, in $r(x + 1) = r(x) + 2$), the equation has the form $r(x + 1) = G(r(x))$. In this case the function $r(x)$ is defined in terms of the successive iterates of G: $r(1) = G(r(0)), r(2) = G^2(r(0)), \ldots, r(k) = G^k(r, 0), \ldots$.

Iteration may be used as an alternative to minimalization in defining the class of partial recursive functions. This will be described in a later section.