4
Recursive Functions

In this chapter we shall follow one of several possible approaches toward making precise the notion of an effectively computable function. We describe the class of partial recursive functions, a class which may, it is believed, be identified with the vaguely defined class of the (partial) effectively computable functions (cf. Church's thesis, below). We shall, in Chapter 5, where we consider the theory of Turing machines, take still another approach to making sharp the intuitive notion of effective computability. Historically, the formulation of the class of partial recursive functions occurred at about the same time (circa 1936) as the establishment of the theory of Turing machines. These theories, concerned with a mathematical development of the class of computable functions before there were any electronic computers, provided an intellectual orientation that many believe was very helpful in preparing for and motivating the advent of the modern computer.

The operations of composition, recursion, and iteration play key roles in these mathematical formulations, and this suggests their importance in programming. These schemes for defining functions are described here in programming, or in flowchart, terms.

The theory of recursive functions and sets is a very fertile field of mathematics (cf. [9]) and would be so even if no high speed computers existed. We introduce a few of the basic ideas and definitions but leave untouched most of the vast mathematical theory. We define the class of primitive recursive functions, a very important subset of the partial recursive functions. Almost all functions computed in practice are in this class.

At the end of the chapter we give a short description of the basic building blocks used in composing flowcharts for the application of the technique

of structured programming. Our purpose is not to give an exposition of structured programming, but to show that the small number of basic operations used in this disciplined approach to programming provide complete generality. These operations are sufficient to enable the execution of any computable procedure whatsoever. The proof depends on showing that with these building blocks we are able to compute an arbitrary partial recursive function.

4.1 INTRODUCTION

In considering the underlying mathematics of programming, we are concerned with functions whose domains and ranges are subsets of the natural numbers. (It will be more convenient for us to assume in this chapter that the "natural numbers" include zero. That is, we are now using the term as being equivalent to the "nonnegative integers.") That we are able to dispense with negative numbers and with nonintegral numbers comes as no surprise to anyone with the briefest acquaintance with computers. For, as we have emphasized earlier when considering number representation in the computer, every signed number that is represented within the machine—floating or fixed point—can always be thought of as an integer (the integer corresponding to the string of bits that is used to represent the number in, say, a binary machine). We shall usually assume domains and ranges to be subsets of the natural numbers, but we shall also have occasion to introduce functions defined for more than a single argument, that is, functions whose arguments are r-tuples ($r \geq 1$) of the natural numbers and which assume values that are s-tuples ($s \geq 1$) of the natural numbers. Any such function can always be regarded as a set of functions that take on values that are natural numbers (the s functions that give respectively the components of the s-tuples in the range of f).

In *all* uses of a computer, as we have emphasized, no matter how far removed from mathematics a commercial application may seem, we are computing the values of a function. For, in any data-processing operation, the input can always be read off as a string of bits (a natural number) or as an n-tuple of such strings, and the output can similarly be interpreted. All steps intermediate between input and output then comprise the computation of this function.

4.2 "RECURSIVE" DEFINITIONS

In defining the class of recursive functions, we make use of a technique that appears frequently in mathematics in the definition of certain classes of

objects. These classes are defined "recursively" or in terms of themselves. The technique is an example of the genetic methods referred to in Chapter 2 and it consists of the following: we assert first that certain "primitive" objects are in the class being defined. We then describe one or more ways of producing new objects in the class when given some objects already known to be in the class. All members of the class being defined are then taken to be precisely those, and only those, objects that can be obtained by starting with the primitive objects and repeatedly introducing new objects, making use of the stated rules for obtaining these new objects. We have already seen one example of such a technique in the use of recursion in the definition of a function. In this case, the method enables us to compute the value of $f(n)$ once we have obtained the value of $f(n-1)$. (The "objects" being defined are the elements $(n, f(n))$ of the graph of $f(n)$.)

For example, consider the class P of numbers defined in the following way.

Primitive object: 1 is a member of P.

Generating rule: If x is a member of P and y is a member of P (not necessarily distinct from x) then $x + y$ is a member of P.

Very little thought shows that the class P consists of all the positive integers.

In formal logical systems the same procedure is used in defining a class of well-formed formulas (wff's) starting with certain primitive formulas and rules which describe how to formulate new wff's from given wff's. We have seen the technique illustrated in the case of Boolean algebra.

We may, for example, define the class of well-formed parenthetical expressions (PE's) formed from two symbols a, b by writing:

Primitive parenthetical expressions: a and b are well-formed parenthetical expressions.

Generating rules: (1) If E_1, E_2 (not necessarily distinct) are well-formed parenthetical expressions, then (E_1E_2) is a well-formed parenthetical expression. (2) If E_1 is a well-formed parenthetical expression, then (E_1) is a well-formed parenthetical expression.

This scheme admits the following expressions as well-formed. We generate these expressions by several applications of the given rules. The expression on the right-hand side of an arrow is obtained from expressions on the left of the arrow by applying the named rule.

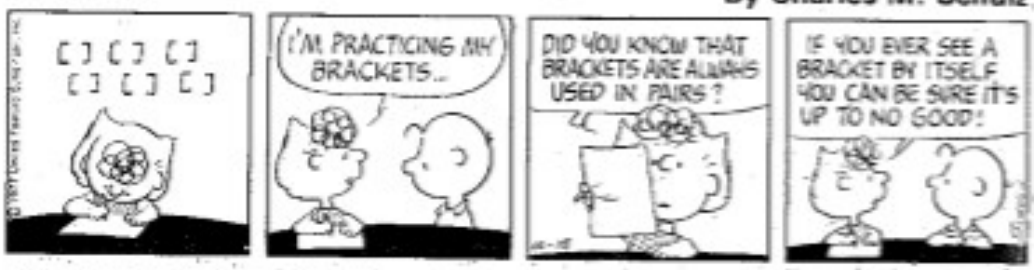
Expression	*Generation*					
$((ab)b)$	a, b	Rule 1 $\rightarrow$	(ab)	Rule 1 $\rightarrow$	$((ab)b)$	
$(((b(ab))))$	a, b	Rule 1 $\rightarrow$	(ab)	Rule 1 $\rightarrow$	$(b(ab))$	
		Rule 2 $\rightarrow$	$((b(ab)))$	Rule 2 $\rightarrow$	$(((b(ab))))$	

The elegant notational scheme, the Backus-Naur form, which was introduced in Chapter 2, is often used in describing the syntax of programming languages where recursive definitions of symbolic expressions are applied (see Chapter 9). Its use is illustrated in the following application to the last example:

$$<PE> ::= a \mid b \mid (<PE>) \mid (<PE><PE>)$$

This means that the class of well-formed parenthetical expressions, PE, can be formed by including a or b or any PE surrounded by parentheses or any juxtaposition of PE's surrounded by parentheses.

PEANUTS **By Charles M. Schulz**



We now make use of these methods to define several classes of functions. The first step is to define the primitive objects, or basic functions, of these classes of functions.

4.3 THE BASIC RECURSIVE FUNCTIONS

The following three functions are those with which we begin the generation of the classes of the primitive recursive functions and of the partial recursive functions. As might be expected, these three basic functions are, in some sense, the simplest functions that we can consider. We follow the development and notation as in Davis [3].

1. The successor function, $S(x) = x + 1$. For given argument x, the image of x is its successor $x + 1$.
2. The zero function, $N(x) = 0$. For any given argument x, this function is identically zero.
3. The generalized identity functions defined for all i, n, with $0 < i \leq n$, $U_i^{(n)}(x_1, \ldots, x_n) = x_i$. For a given n-tuple of arguments, this function is the ith component of the n-tuple.

These functions are the building blocks with which we start in generating the very rich classes we now define. It is seen that they are indeed primitive functions.

Generating Rules

Composition. Suppose we know that a function $f(y_1, \ldots, y_m)$ is in the class being generated. We can apply the composition rule to yield a new function by replacing each of the arguments y_i $(i = 1, 2, \ldots, m)$ by any function that has been shown to be within the class.

Thus, in formal terms, if we are given the functions

$$\begin{cases} f(y_1, \ldots, y_m) \\ g_1(x_1, \ldots, x_n) \\ g_2(x_1, \ldots, x_n) \\ \quad \vdots \\ g_m(x_1, \ldots, x_n) \end{cases}$$

as members of the class being generated, then the function

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

is obtained from these given functions by the operation of composition (cf. Fig. 4.1).

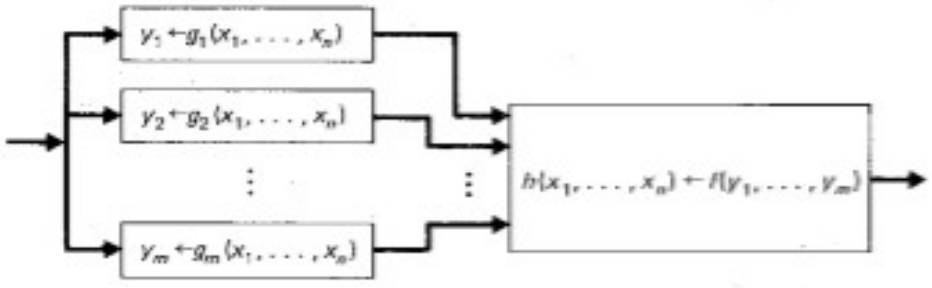


Fig. 4.1 The operation of composition-computation of $h(x_1, \ldots, x_n)$.

For example, the function $F(x) = 1$ is obtained by substituting $N(x)$ for x in $S(x)$ as follows:

$$F(x) = S(N(x)) = N(x) + 1 = 1.$$

The function $G(x) = 2$ can then be obtained by

$$G(x) = S(F(x)) = F(x) + 1 = 2.$$

The function $\varphi(x_1, x_2) = 0$ is obtained by the following composition scheme:

$$\varphi(x_1, x_2) = N(U_1^{(2)}(x_1, x_2)) = N(x_1) = 0.$$

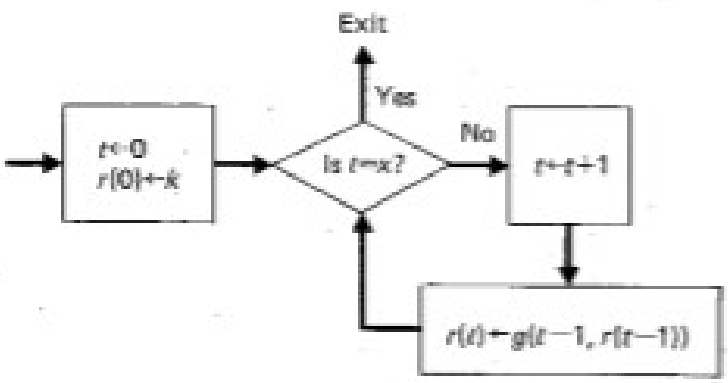
The function $f(x, y) = x + 2$, which is not the same as the function $\varphi(x) = x + 2$, is derived as follows:

$$f(x, y) = S(S(U_1^{(2)}(x, y))) = x + 2.$$

Using composition alone, a rather restricted class of functions can be generated from the three basic functions. Note that the SUBROUTINE operation and the MACRO facilities of a number of programming languages provide obvious facilities to use composition within a program.

Primitive Recursion. The rule or operation of primitive recursion is a schema (Fig. 4.2) that defines a new function in terms of two given total functions. We show first how this schema defines a function of a single argument. Call the function to be defined $r(x)$. The operation of primitive recursion includes an assignment of some specified value to $r(0)$. It then also includes a procedure to determine $r(k + 1)$, once we have obtained $r(k)$. We write the following equations, where k is a constant and $g(x, y)$ is assumed to be a total function already known to be within the class.

$$\begin{cases} r(0) = k \\ r(x + 1) = g(x, r(x)) \text{ for } x \geq 0 \end{cases}$$



Thus, starting with $r(0)$, we are able to iterate the second equation to obtain r at any argument.

More generally, we can define a function, $r(x_1, \ldots, x_n)$, of a number of arguments by applying this idea to some one of the arguments. Thus the following equations define $r(x_1, x_2, \ldots, x_n)$ by (1) giving its value for $x_1 = 0$ and arbitrary values of the other arguments, and (2) defining r with first argument $x_1 + 1$ in terms of the value of r with first argument x_1.

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

The function $r(x_1, \ldots, x_n)$ has been defined in terms of the functions $f(x_2, \ldots, x_n)$, $g(x_1, y, x_2, \ldots, x_n)$.

4.4 POSSIBLE PROGRAMMING LANGUAGE FEATURES TO ENABLE THE USE OF FUNCTION-GENERATING OPERATIONS

The operations of composition and of primitive recursion are examples of transformations of functions into functions, as we have discussed this in Chapter 2. They are mappings of a higher order than ordinary functions that map numbers into numbers. In both composition and primitive recursion the input for each operation consists of *functions*, and the output is a *function*. As we have pointed out, most programming languages have facilities to define ordinary functions, i.e., to define procedures whose arguments are variables that are assigned specific numeric values at execution time. After the procedure is defined, a program statement provides these input arguments in a call statement such as, typically

```
Z = FUNCTION(X1, X2);
```

The output value of the function is assigned to the variable Z when this statement is executed.

However, we need some additional language capability, provided by only a few languages such as ALGOL 68, in order to be able to cope with the definition of the operations of composition or of primitive recursion. Here, we might want to be able to write, as part of a procedure *definition*, statements like:

```
H(X1,X2) = COMPOS(F(Y1,Y2,Y3),G1(X1,X2),G2(X1,X2),G3(X1,X2));
```

or

```
R(X) = PRIMREC (K, G(X, Y));
```

These refer respectively to special cases of the operations of composition and of primitive recursion. We have used the upper case equivalents of the identifiers that appeared in our original definitions of these operations.

After defining the functions H(X1, X2), R(X), we should, as before, l able to call them in statements such as:

$$Z = H(X1, X2); \text{ or}$$
$$Z = R(X);$$

The procedures COMPOS and PRIMREC that appear in these proc dure definitions use *functions* as their arguments and give as a result *function*, not a number. In calling COMPOS and PRIMREC the input a guments are other *procedures*, not variables. This is another example of tl kind of situation that justifies the extended capability for procedure defi tions of ALGOL 68.

We shall make use later of such programming language facilities describe the generation of functions in the class of partial recursive fun tions.