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.

4.11 PRIMITIVE RECURSION AND MATHEMATICAL INDUCTION

The method of mathematical induction for proving theorems is closely related to the operation of primitive recursion. Induction is often used to prove a theorem $T(n)$ that is stated in terms of a parameter n, where n is an arbitrary natural number (e.g., "The sum of the first n positive integers is $n(n+1)/2$"). The method, which most people have learned in secondary school or introductory college algebra courses, consists of proving a stated theorem to be true for $n = 0$ (or perhaps $n = 1$) and then showing that *if* the theorem is true for $n = k$ (i.e., assume $T(k)$ to be a true statement), then the theorem must be true for $n = k + 1$. That is, (i) $T(0)$ is true, and (ii) $T(k)$ implies $T(k+1)$. The two parts imply that $T(n)$ is true for all integers n (≥ 0 if we use 0 in the first part, ≥ 1 if we start with 1). We recall the use of the method in the following.

Example. Prove that the sum $S(n)$ of the first n squares is $S(n) = n(n+1)(2n+1)/6$.

i. For $n = 1$, $S(1) = 1$, and the theorem is seen to be true for this initial value of n.
ii. Assume that $S(k) = k(k+1)(2k+1)/6$. We must prove that $S(k+1)$ is given by the value of $n(n+1)(2n+1)/6$ at $n = k+1$.
 We have

$$\begin{aligned} S(k+1) &= \sum_{i=1}^{k+1} i^2 = \sum_{i=1}^{k} i^2 + (k+1)^2 \\ &= S(k) + (k+1)^2 = k(k+1)(2k+1)/6 + (k+1)^2 \\ &= (2k^3 + 9k^2 + 13k + 6)/6 = (k+1)(k+2)(2k+3)/6, \end{aligned}$$

and the proof by induction is complete.

Example. If $f(m, n)$ is the Ackermann function, prove that $f(1, n) = n + 2$.

i. For $n = 0$, $f(1, 0) = f(0, 1) = 2$ and, thus, the given statement is true for $n = 0$.

ii. Assume that $f(1, k) = k + 2$. We must show that

$$f(1, k + 1) = (k + 1) + 2 = k + 3.$$

We have $f(1, k + 1) = f(0, f(1, k)) = f(0, k + 2) = k + 3$. The stated result is then shown to be true.

Using the result that $f(1, n) = n + 2$, we leave it to the reader to show by induction that $f(2, n) = 2n + 3$.

It can be similarly shown that:

$$f(3, n) = 2^{n+3} - 3$$

$$f(4, n) = \left.{2^{2^{\cdot^{\cdot^{2}}}}}\right\}{\scriptstyle n+3} - 3$$

(Note that the first two results can be written to conform with this pattern if we write:

$$f(2, n) = 2(n + 3) - 3$$
$$f(1, n) = 2 + (n + 3) - 3.)$$

The operation of primitive recursion provides the definition of a computable function $f(x)$ when we are given its value at $x = 0$ and have a means of computing $f(x)$ at $k + 1$ if we know the value of $f(x)$ at k. Mathematical induction provides the proof of a theorem involving n when we can prove the theorem to be true at $n = 0$ and can prove that the theorem is true at $n = k + 1$ if it is true at $n = k$.

4.12 THE USE OF ITERATION AS AN ALTERNATIVE TO MINIMALIZATION[1]

A variant of the operation of minimalization that can serve equally well in the definition of the class of partial recursive functions is that of iteration as this has been used by Eilenberg and Elgot [7].

Iteration. Let $f(x)$, $g(x)$ be two partial functions known to have distinct domains (that is, they are never both defined for the same arguments). We form a new function $h(x)$ as follows.

Given an argument x (here, x may represent an n-tuple $(x_1, x_2, \ldots, x_n)$), either f or g or neither is defined at x.

1. This section can be omitted on a first reading of this material. It covers a topic not normally included in an elementary treatment of this subject area and probably demands a higher level of mathematical sophistication than most of the topics we discuss. We include it because it will facilitate a proof in Section 5.10.

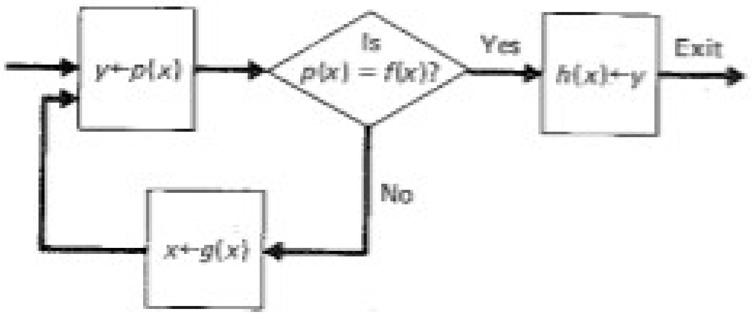


Fig. 4.6 The Eilenberg—Elgot variant of iteration.

Let $f \cup g = p$ denote the function which for a given argument x is equal to $f(x)$ if $f(x)$ is defined, or is equal to $g(x)$ if $g(x)$ is defined. Otherwise, p is undefined at x. We are assured that the definition is unambiguous since f, g have distinct domains. It can be shown, but it is not immediate, that $p(x)$ is partial recursive if $f(x)$, $g(x)$ are partial recursive.

We then define $h(x)$ as follows:

1. If $p(x)$ is defined at x, then if $p(x) = f(x)$ go to (2), else if $p(x) = g(x)$ then go to (3); if $p(x)$ is not defined at x then neither is $h(x)$.
2. $h(x) = f(x)$.
3. Replace x by $g(x)$ and return to (1).

The function $h(x)$ can also be written as $h = f \cup fg \cup fg^2 \cup \ldots \equiv fg^*$. ($g^k$ denotes the function $\overbrace{g(g(\ldots(g(x))\ldots))}^{k\ g\text{'s}}$. As noted earlier, it is called the kth iterate of $g(x)$.) $h(x)$ is said to be obtained by iteration from $f(x)$, $g(x)$.

Consider the problem of programming the computation of $p(x)$ with $f(x)$, $g(x)$ being partially computable and not total. If $p(x)$ is defined we must avoid an endless computation of either $f(x)$ or $g(x)$. We can accomplish this by proceeding as follows. Assume that we have procedures available to compute $f(x)$ and $g(x)$. Let these be conducted in parallel, that is, take one or several steps in the computation of $f(x)$, then do the same for $g(x)$ and continue to alternate back and forth between the two procedures for $f(x)$ and $g(x)$ until one or the other terminates. If either procedure terminates, yielding a value of $f(x)$ or of $g(x)$, continue the computation of $h(x)$ with step (2) or with step (3). We are assured that it is not possible for both procedures (for $f(x)$ and for $g(x)$) to terminate for the same argument since the ambiguous situations where both $f(x)$ and $g(x)$ are defined cannot occur.

If a function is obtainable by the operation of minimalization, it is obtainable through the use of iteration. This is quickly seen from the following.

Suppose that $h(x) = \min y[f(y, x) = 0]$ where $f(y, x)$ is a total function of x, y.

Let the function $\varphi(x, y)$ be defined as follows:

$$\varphi(x, y) = \begin{cases} y \text{ if } f(y, x) = 0 \\ \text{undefined if } f(y, x) \neq 0 \end{cases}$$

Let $g(x, y)$ be defined as:

$$g(x, y) = \begin{cases} (x, y + 1) \text{ if } f(y, x) \neq 0 \\ \text{undefined if } f(y, x) = 0 \end{cases}$$

(The range of $g(x, y)$ consists of a set of ordered pairs.)

Thus, $\varphi(x, y)$ and $g(x, y)$ have distinct (complementary) domains. We leave it to the reader to verify that we can then write

$$h(x) = (\varphi \cup \varphi g \cup \varphi g^2 \cup \ldots)(x, 0) = (\varphi g^*)(x, 0).$$

The flowchart of Fig. 4.7 shows the computation of $\varphi g^*(x, 0)$ or, equivalently, of $h(x)$.

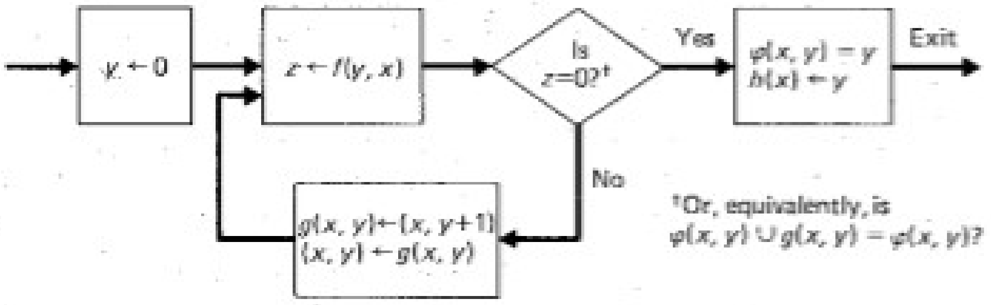

$y \leftarrow 0$
$z \leftarrow f(y, x)$
Is $z = 0$?†
Yes
$\varphi(x, y) = y$
$h(x) \leftarrow y$
Exit
No
$g(x, y) \leftarrow (x, y+1)$
$(x, y) \leftarrow g(x, y)$
†Or, equivalently, is $\varphi(x, y) \cup g(x, y) = \varphi(x, y)$?

Fig. 4.7 Minimalization viewed as an example of iteration (of the function g).

We can similarly prove the converse, that a function defined by iteration can also be defined by minimalization, but a complete proof will require extensive details (see [7]).

4.13 ON SETS: PRIMITIVE RECURSIVE, RECURSIVE, RECURSIVELY ENUMERABLE

A set is a collection, finite or infinite, of natural numbers or of n-tuples of the natural numbers. We are thus, as noted in Chapter 3, using the word with a more restricted meaning than it has in many parts of mathematics.

Associated with every set, as previously noted, is its characteristic func-tion, that function which for a given argument assumes the value 0 if tha argument is in the set, and 1 if it is not. A characteristic function is, then necessarily total. If the characteristic function is primitive recursive or re-cursive, we say that the set is, respectively, primitive recursive or recursive

A larger class of sets is that of the *recursively enumerable* sets. These sets have a generative structure in the sense that each recursively enumerable set is the set of outputs of some computable function.

Definition: *A set is recursively enumerable if it is empty or is the range o some recursive function.*

Note the difference between a recursive set and a recursively enumer-able set. Since the characteristic function is recursive or computable when the set is recursive, we are able, given a number, to determine by computing its characteristic function whether or not that argument is in the set. When it is recursively enumerable we may not be able to make such a determination.

In other words, if a set is recursive we can program a computer so that, given an arbitrary argument, the programmed machine will arrive at some conclusion as to whether or not the given argument is in the set. It may be necessary to wait a very long time to make this determination, but we are assured that if we wait long enough a response will be forthcoming as to whether or not the given argument is a member of the prescribed recursive set.

The set of prime numbers is, for example, recursive, for we can easily describe a procedure that will determine if a given argument is prime or not. We can, in a very straightforward manner, divide the given number by 2, 3, 4, ... trying all numbers up to its square root (that we need not try *all* divisions, but can restrict our inquiry to prime divisors is an obvious saving in efficiency—but we are interested only in the feasibility of the determina-tion procedure). If our search yields no divisors of the given number, that number is, of course, prime.

In the case of a recursively enumerable set we can program a machine to compute successively the members of the set $\{f(0), f(1), f(2), \ldots\}$, and, even-tually, every element of the set will appear in its output. However, given an arbitrary number, we cannot, as in the case of a recursive set, be certain of being able to make a determination of whether or not that number is in the set. We may wait a very long time and, perhaps, have our question answered affirmatively by seeing it appear as an output of the computation. However, if the argument is *not* in the set, this negative answer never appears. No matter how long we wait we are never able to abandon our search with assurance that the sought-for argument will not appear in subsequent calculations of the function. In other words, there is a *partial* determination procedure available to answer our question. If the answer is YES, we eventually learn this, but if the answer is NO we can never realize it.

A recursive set is necessarily recursively enumerable. To see this, note that if we have a computational procedure to determine if a given argument is in a particular recursive set, we can modify the procedure so that instead of stopping and giving the answer YES when the argument is in the set, it gives as final result the input argument itself. Also, in those instances where the argument is not in the given set, the procedure is modified to give as output some particular element of the set, and by our assumption there is at least one. The range, then, of the function computed by this procedure is the given recursive set and, therefore, this set is recursively enumerable.

It is not at all easy to give an example of a set that is known to be recursively enumerable but is not recursive. We shall see an example of such a set in Chapter 6 (on the theory of Turing machines). We can perhaps, however, make plausible the fact that being a recursive set is a more demanding condition than being recursively enumerable.

Consider the following function. Let $\psi(x)$ be the smallest number of prime numbers into which x can be partitioned (i.e., the smallest number of prime numbers whose sum is x). The range of this function, since it is obviously programmable, is by definition a recursively enumerable set. Call it S. A famous problem in elementary mathematics, namely Goldbach's conjecture, concerns the question of whether every even number can be expressed as the sum of two primes. It was first posed in 1742, but no one has yet succeeded in solving it completely. Thus, if this conjecture is true, $\psi(2x)$ is equal to 2 for all x (> 1), and S consists of only the number 2.

The range S of $\psi(2x)$ is, by definition, a recursively enumerable set since ψ is a recursive function. The proof that ψ is recursive involves a fair amount of detail if we attempt to show this by generating it using composition and minimalization starting with the three basic recursive functions. However, if we accept Church's thesis, we can argue that since $\psi(2x)$ is obviously a programmable function, then it is recursive.

The range of $\psi(2x)$ is then recursively enumerable, but is it a recursive set? In order to show this, we would have to prove that there is a recursive function which can be used to determine whether or not a given argument is in the range. We do not know how to define such a function. It has been proven by great effort that every even number can be partitioned into not more than 68 prime numbers and, therefore, that no number greater than 68 is in S. However, no one knows at this writing if, for example, 3 is in S and certainly there is no known recursive function that can be used to make this determination for 3 or for any other argument.

The possible—in the sense of conforming to syntactical rules of wel
formedness—programs that can be written in any well-defined programmin
language form a recursively enumerable set. We offer a heuristic argumen
for this fact. One could program a machine to generate all possible program
with 1 instruction—with 2 instructions— . . . and so forth. This set of program
is also recursive; that is, there is a well-defined procedure (effective compu
tation) to determine if a given program is well-formed (see Chapter 9). For, i
the program has n instructions, one extravagantly inefficient way of doin
this would be to generate all well-formed programs with n instructions an
scan this list to determine if the given program is among them.

The set of all theorems in a formal system constitutes a recursivel
enumerable set (note that here we have a set of character strings, not o
numbers). This can be seen by the fact, accepting Church's thesis, that w
should be able to describe a mechanical procedure that will churn out al
possible theorems. This, under rather general circumstances, can be accom-
plished by describing a procedure that will go about applying the generatin
rules, or rules of derivation, starting with the axioms, in all possible ways

Every reader has studied elementary Euclidean plane geometry and, a
noted earlier, has, therefore, seen a prototype of a formal system. However, a
good deal of further "formalization" would have to be done before thi
subject, as it is usually described, would lend itself to such a mechanical
theorem-proving procedure. A clearer example for our purpose would be the
axiomatic formulation of the propositional calculus, as we have described
this in Chapter 2. This schema included four axioms and two rules (substi-
tution and modus ponens) for generating new theorems. It is feasible to write
a program that defines a procedure to go about applying these rules to the
axioms in all possible ways, generating eventually all possible theorems.

We shall only in the most sketchy way try to make this result credible
and generally applicable to formal systems. Thus,

$$A_1, A_3, A_4, G_1(A_1, A_3), G_2(A_3, A_4, G_1(A_1, A_3))$$

might describe the generation of a theorem starting with given axioms
A_1, A_3, A_4, and applying certain generating rules G_1, G_2 which take respec-
tively two arguments and three arguments. It will take a lot of work to
implement a general procedure of this kind—including substitutions of new
arguments for the variables that may appear in the generation scheme—but
it can be done.

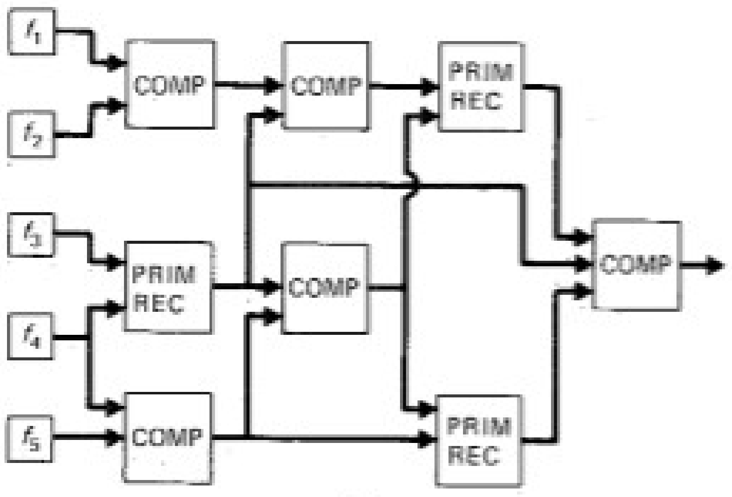
It is easy to show that any *finite* set of numbers is recursive. All we have
to do to program a machine to recognize whether or not a given number is in
the set is to store within the machine the numbers in the given set and
compare a given argument with each of these stored numbers.

4.14 ON STRUCTURED PROGRAMMING

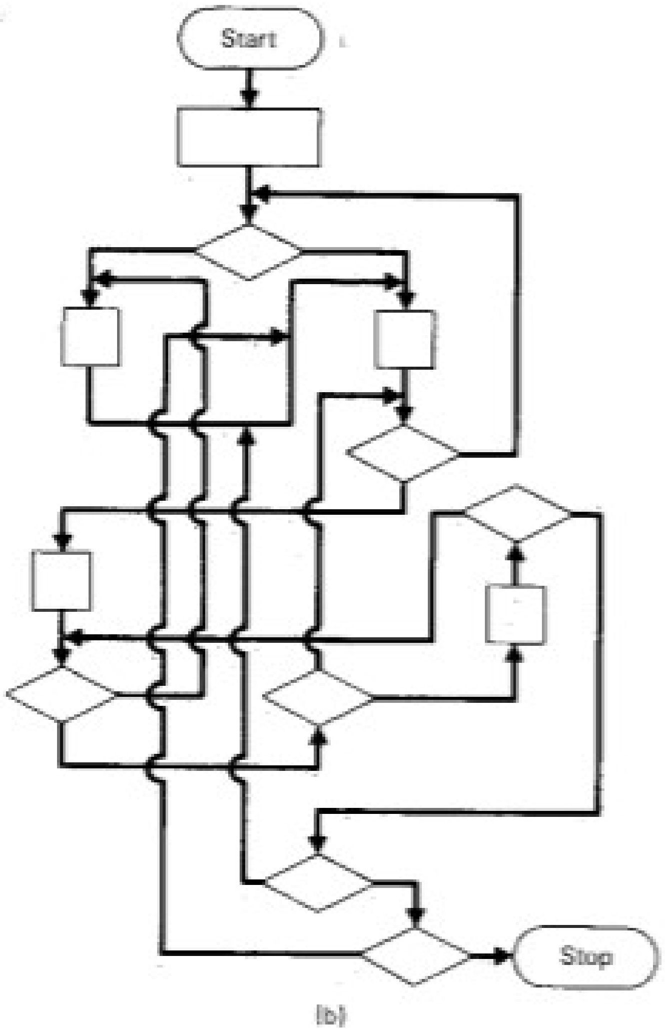
We shall touch briefly on the mathematical ideas that underlie the technique of structured programming, a discipline for writing programs that has been recently advocated, principally by Dijkstra (cf. [2, 4, 5, 6, 8]), in order to facilitate program development and to minimize programming errors. One key point is the argument [4] that the unbridled use of GO TO's or TRANSFER's in programming leads often to a program with an excessively complex and tortured logical flow, and this makes errors likely and debugging difficult. We have, however, seen that a rather elegant scheme is available to describe the generation of all (accepting Church's thesis) computable functions. That is, we can describe the computation of every such function by starting with a very small number of primitive functions and by cascading, in proper sequence, the repeated application of a very few fundamental operations—composition and minimalization, or composition and iteration. We have also seen that composition and primitive recursion suffice for almost all computable functions that occur in practice.

In looking at the flowchart for the generation of a function such as in Fig. 4.8(a), we do not see an intricate, tortured flow of the kind that might appear in a program that made use of different kinds of atomic steps, especially GO TO operations (Fig. 4.8(b)). Note, in fact, that there are *fewer* processing blocks in Fig. 4.8(b), but Fig. 4.8(a) seems less complicated. In particular, with no GO TO's, the flow is straightforward; the intermediate functions needed in the process are developed using one or the other of the basic operations. Finally, the output function is realized after the necessary preliminary functions have been generated. All of this seems to lend credence to the fact that we can avoid unduly contrived situations in planning program development. We shall not, however, use the schema for the generation of the recursive functions.

It is the idea of repeatedly applying a few simple operations that is exploited in the method of structured programming. This technique is, perhaps, best described as a method for composing, by successive refinements, flowcharts that make use of a very small number of basic building blocks. Flowcharts appear in computing in many ways, often with unclearly specified conventions. In particular, when a directed line joins two parts of a flowchart it may, on occasion, denote a sequencing of events, or a moving of data, or both. We shall assume in the following that a directed line joining two parts of a flowchart indicates not necessarily a moving of data but rather specifies an ordering in time of the sequence of steps involved in the execution of the process. Thus, a self-contained segment of a flowchart will have just one input line and one output line. We shall allow lines to merge, but not split (except through decision boxes).



(a)

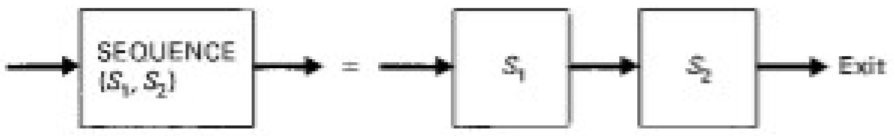


(b)

Fig. 4.8 Examples of flowcharts for (a) the generation of a primitive recursive function, (b) a procedure with unrestricted use of transfer instructions.

Three basic building blocks are used in composing structured flowcharts. These are:

1. the sequencing of two processes:



2. the conditional test or the "if then else":

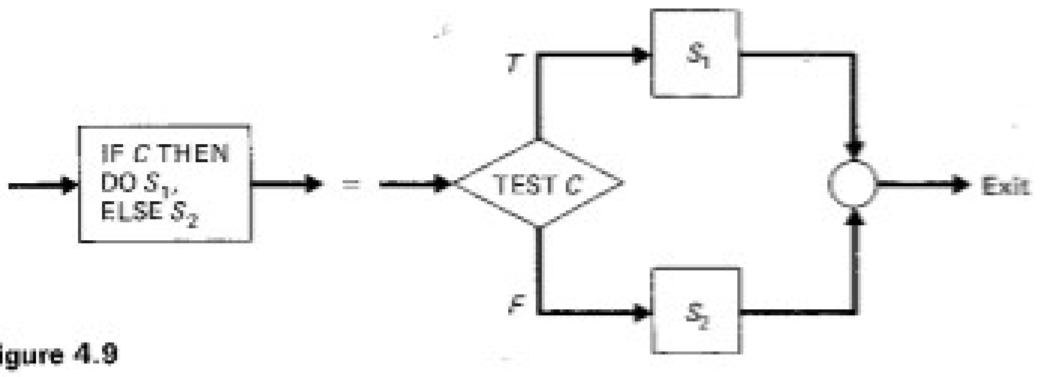


Figure 4.9

3. the "do while":

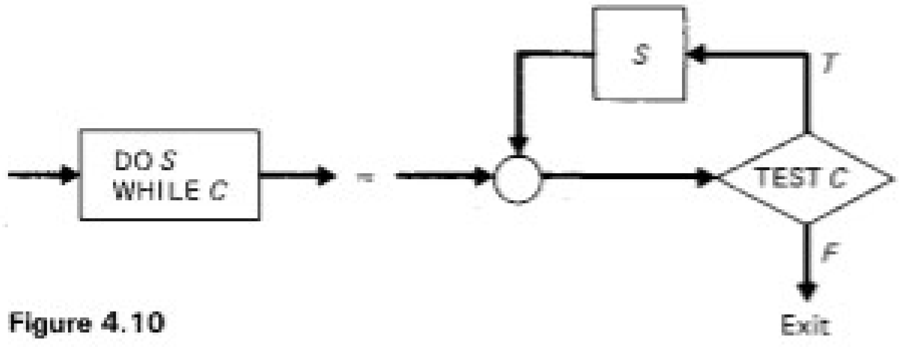


Figure 4.10

Since each of the three building blocks or elements has one input line and one exit line, we can describe in recursive terms the generation of complex flowchart structures from these elements. Every block that denotes a process (the rectangular blocks in the preceding) can be replaced in turn by one of the building blocks, or more generally by a flowchart with one input and one output. (We leave it to the reader to describe this class of flowcharts by an extended BNF that applies to flowcharts in a manner analogous to th application of the BNF notation to sequences of symbols.) The primitiv processes in this development can be taken to be the three basic function $N(x) = 0$, $S(x) = x + 1$, $U_i^{(n)}(x_1, x_2, \ldots, x_n) \doteq x_i$, and the primitive condi tion can be taken as $x = y$.

The adequacy of the three building blocks to express any computin process whatsoever is demonstrated if we show how to perform the fun damental operations of recursive function theory using them.

Composition (Fig. 4.1) is a direct example of the "sequence" operatio and it is so described in Fig. 4.11. Minimalization (Fig. 4.5) is easily de scribed in terms of the "do while" operation as we see in Fig. 4.12.

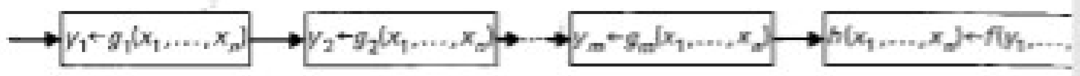

$y_1 \leftarrow g_1(x_1, \ldots, x_n)$
$y_2 \leftarrow g_2(x_1, \ldots, x_n)$
$y_m \leftarrow g_m(x_1, \ldots, x_n)$
$h(x_1, \ldots, x_n) \leftarrow f(y_1, \ldots,$

Fig. 4.11 Composition in terms of the "sequence" operation

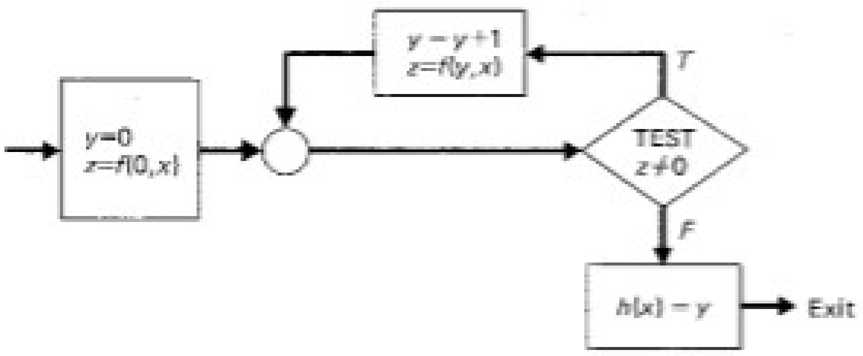


Fig. 4.12 Minimalization using the "do while."

Primitive recursion (Fig. 4.13) and iteration can similarly be easily ex pressed in terms of the basic building blocks.

We have defined both minimalization and primitive recursion in term of the "do while." However, pursuant to our earlier remarks, note the differ ence in its use in these two cases. In primitive recursion an ordinary do loo can be used in program execution. We know at the entry to the loop that it i to be executed x times. In the case of minimalization nothing is known i general, at the entry to the loop, of the number of times the loop will b executed. In the "do while" search for a zero of $f(y, x)$ we are not eve

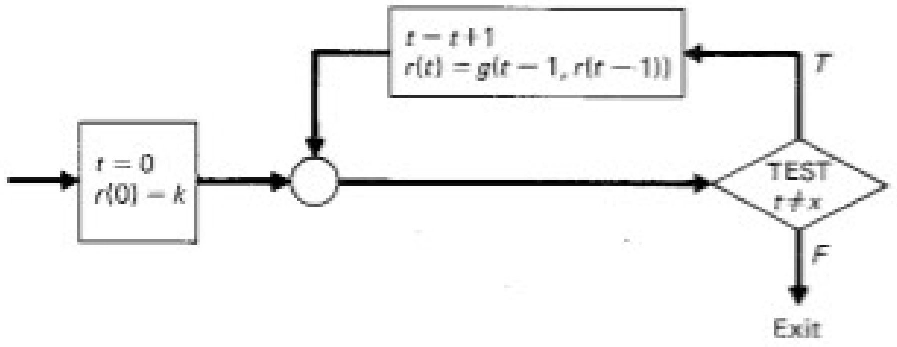


Fig. 4.13 Primitive recursion (defining a function, $r(x)$, of a single argument) using the "do while."

assured in the general case that the search will eventually terminate (the function being computed may not be total).

Thus, there is no question of the universal applicability of these building blocks. The technique of structured programming is then applied by first describing an algorithm and using processing blocks in the flowchart with rather sweeping functional specifications that indicate what the program is to accomplish. Then, by successive refinements the processing blocks are replaced by nested equivalent flowcharts of greater and greater detail until the implementation of the program becomes routine.

EXERCISES

4.1 Show that the following functions of nonnegative integers are primitive recursive:

a. $f(x) = (3x^2 + 5x) \div 2$

b. $g(n) = 10^n$

c. $\begin{cases} F(n) = 1^3 + 2^3 + 3^3 + \ldots + n^3 \text{ for } n \geq 1 \\ F(0) = 0 \end{cases}$

4.2 If x, y are nonnegative integers and x is divided by y ($\neq 0$), we get a quotient $Q(x, y)$ and a remainder $R(x, y)$, where $R(x, y) < y$, satisfying the equation

$$\frac{x}{y} = Q(x, y) + \frac{R(x, y)}{y}.$$

Show that both $Q(x, y)$ and $R(x, y)$ are recursive functions. (Show that each can be obtained by composition and/or minimalization from functions already shown to be primitive recursive.)

4.3 Assume that a PL/I (or other suitable programming language) subroutine FUNCT(X, Y) exists. Write a segment of code that defines a new function subroutine, $H(X)$, defined by:

$$H(X) = \min Y \text{ (FUNCT } (X, Y) = 0).$$

4.4 Show that the functions $p = \max(x, y)$, $q = \min(x, y)$ are primitive recursive.

4.5 Let $f(x)$ be defined by:

$$f(x) = \begin{cases} x^2 \text{ if } x \text{ is even} \\ x + 1 \text{ if } x \text{ is odd} \end{cases}$$

Show that $f(x)$ is primitive recursive.

4.6 Show that if $g(x_1, x_2)$ is primitive recursive and if $f(x_1, x_2) = g(x_2, x_1)$ then $f(x_1, x_2)$ is also primitive recursive.

4.7 a. Show that the function $y = [\sqrt{x}]$ (the greatest integer contained in $\sqrt{x}$) is primitive recursive.

b. Show that $y = [x/3]$ is primitive recursive.

4.8 Describe the class of functions that is generated by starting with the three basic functions: $S(x)$, $N(x)$, $U_i^{(n)}(x_1, x_2, \ldots, x_n)$ and using repeatedly only the operation of composition in all possible ways.

4.9 If $f_0, f_1, f_2, \ldots$ is an enumeration of all singulary (one argument) primitive recursive functions, prove that the binary function $f_x(y)$ of the arguments x, y is *not* primitive recursive.

4.10 Let $f(x)$ be defined by the equations:

$$\begin{cases} f(0) = 1 \\ f(1) = 3 \\ f(x + 2) = (x + 2) \cdot f(x) \text{ for } x > 1 \end{cases}$$

Show that $f(x)$ is recursive.

4.11 If $f(m, n)$ denotes the Ackermann function, compute $f(2, 2)$ by following the flowchart of Fig. 4.4.

4.12 We have proved by induction that $f(1, n) = n + 2$ where $f(m, n)$ is the Ackermann function. Show, similarly, that:

a. $f(2, n) = 2n + 3$

b. $f(3, n) = 2^{n+3} - 3$

c. $f(4, n) = \left.2^{2^{\cdot^{\cdot^{2}}}}\right\}{\scriptstyle n+3} - 3$

4.13 If $f(m, n)$ is the Ackermann function, show that the function $f(x, x)$ is not primitive recursive.

Outline of proof: Let $\varphi(x_1, x_2, \ldots, x_n)$ be an arbitrary primitive recursive function. Show first:

A: There is some number M, depending only on φ, such that

$$\varphi(x_1, x_2, \ldots, x_n) < f(M, \max(x_1, x_2, \ldots, x_n)) \text{ for all } x_1, x_2, \ldots, x_n.$$

Prove A by showing: (1) it is true for the three basic primitive recursive functions $S(x)$, $N(x)$, $U_i^{(n)}(x_1, x_2, \ldots, x_n)$, (2) $f(m, n)$ is increasing in each argument, (3) if a set of functions satisfy property A, then any function formed from them by composition or primitive recursion satisfies A.

Thus, if $\varphi(x)$ is an arbitrary primitive recursive function of a single argument there is some number M such that $\varphi(x) < f(M, x)$ for all x. If $f(x, x)$ is primitive recursive, we can let $\varphi(x)$ be $f(x, x)$ in this inequality. Letting $x = M$, we have a contradiction, and therefore $f(x, x)$ is not primitive recursive.

4.14 Show that if $f(x)$ is a recursive function satisfying $f(n) < f(n + 1)$, then the range of $f(x)$ is a recursive set.

4.15 Assume a stored program computer with only one arithmetic register, an accumulator, and only the following three machine operations:

TR + A:	Transfer to location A if the contents of the accumulator are positive.
ADD A:	Add the contents of location A to the contents of the accumulator.
STO A:	Store the contents of the accumulator, without clearing it, in location A.

Suppose that there is an unlimited store in the machine (or at least that it is as big as we need for any given application), and that the word size is also as big as may be needed. Assume, also, that any needed constants like $0, +1, -1$ are stored permanently in certain fixed locations.

Show that, accepting Church's thesis, this very limited instruction repertoire is adequate to perform any computation whatsoever. A function of N arguments is computed by setting up its arguments in, say, locations $1, 2, 3, \ldots, N$ and putting the result in location $N + 1$. It suffices to prove that the three basic functions can be computed on this machine, and that functions obtained by composition, primitive recursion, minimalization from machine computable functions are, themselves, machine computable.

4.16 Write a program in any suitable language that will generate and emit as output all the prime numbers. Assume that the word size of the machine is unlimited. This result, then, implies that the prime numbers form a recursively enumerable set.

4.17 Write a program in any suitable language that will generate and emit as output all possible ordered pairs of the natural numbers. Assume that the word size is unlimited.

4.18 Write a program in any suitable language that will generate and emit as output all possible programs for the machine of Exercise 4.15, i.e., all possible programs formed from the operations TR + , ADD, STO. Again, assume an unlimited word size.

4.19 Show that the operation of iteration (as described by Fig. 4.6) can be obtained from the "sequence," "if then else," and "do while" operations of structured programming.

4.20 Show that the "sequence," "if then else," and "do while" operations of stru tured programming are describable in terms of the operations of compositio primitive recursion, and minimalization.

4.21 Describe, as implied possible in the text, the class of flowcharts formed fro the basic building blocks of structured programming using an extended BNF appl cable to flowcharts.

4.22 Show that the iterative algorithm for the computation of $\sqrt{N}$ approximatel doubles, with each iterate, the number of correct decimal places in the approxima tion to $\sqrt{N}$.

REFERENCES

1. Ackermann, W., "On Hilbert's Construction of the Real Numbers," 192 Translated by Stefan Bauer-Mengelberg in *From Frege to Gödel.* Edited by Jea van Heijenoort. Cambridge: Harvard University Press, 1967.
2. Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* Ne York: Academic Press, 1972.
3. Davis, M., *Computability and Unsolvability.* New York: McGraw-Hill, 1958.
4. Dijkstra, E. W., "GO TO Statement Considered Harmful." *Comm. Assn. Comp Mach.* **11,** 3 (March 1968).
5. Dijkstra, E. W., "Structured Programming," in *Software Engineering Tech niques.* Ed. J. N. Burton and B. Randell. NATO Science Committee, 1969; pp 88–94.
6. Dijkstra, E. W., *Notes on Structured Programming.* Technological Universit Eindhoven, The Netherlands, Department of Mathematics, April 1970.
7. Eilenberg, S., and C. C. Elgot, *Recursiveness.* New York: Academic Press, 1970
8. Linger, R. C., H. D. Mills, and B. I. Witt, *Structured Programming.* Reading Massachusetts: Addison-Wesley, 1979.
9. Rogers, H. Jr., *Theory of Recursive Functions and Effective Computability.* New York: McGraw-Hill, 1967.