2

Programs and Computable Functions

1. A Programming Language

Our development of computability theory will be based on a specific programming language $\mathscr{S}$. We will use certain letters as variables whose values are ***numbers.*** (In this book the word ***number*** will always mean nonnegative integer, unless the contrary is specifically stated.) In particular, the letters

$$X_1 \ X_2 \ X_3 \ \cdots$$

will be called the ***input variables*** of $\mathscr{S}$, the letter Y will be called the ***output variable*** of $\mathscr{S}$, and the letters

$$Z_1 \ Z_2 \ Z_3 \ \cdots$$

will be called the ***local variables*** of $\mathscr{S}$. The subscript 1 is often omitted; i.e., X stands for X_1 and Z for Z_1. Unlike the programming languages in actual use, there is no upper limit on the values these variables can assume. Thus from the outset, $\mathscr{S}$ must be regarded as a purely theoretical entity. Nevertheless, readers having programming experience will find working with $\mathscr{S}$ very easy.

In $\mathscr{S}$ we will be able to write "instructions" of various sorts; a "program" of $\mathscr{S}$ will then consist of a ***list*** (i.e., a finite sequence) of

Table 1.1

Instruction	Interpretation
$V \leftarrow V + 1$	Increase by 1 the value of the variable V.
$V \leftarrow V - 1$	If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of V.
IF $V \neq 0$ GOT0 L	If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list.

instructions. For example, for each variable V there will be an instruction:

$$V \leftarrow V + 1$$

A simple example of a program of $\mathscr{S}$ is

$$X \leftarrow X + 1$$
$$X \leftarrow X + 1$$

"Execution" of this program has the effect of increasing the value of X by 2. In addition to variables, we will need "labels." In $\mathscr{S}$ these are

$$A, \quad B_1\ C_1\ D_1\ E_1\ \ A, \ \ B_2\ C_2\ D_2\ E_2\ \ A, \ \cdots.$$

Once again the subscript 1 can be omitted. We give in Table 1.1 a complete list of our instructions. In this list V stands for any variable and L stands for any label.

These instructions will be called the ***increment, decrement,*** and ***conditional branch*** instructions, respectively.

We will use the special convention that ***the output variable Y and the local variables Z_i initially have the value 0.*** We will sometimes indicate the value of a variable by writing it in lowercase italics. Thus x_5 is the value of X_5.

Instructions may or may not have labels. When an instruction is labeled, the label is written to its left in square brackets. For example,

$$[B] \qquad Z \leftarrow Z - 1$$

In order to base computability theory on the language $\mathscr{S}$, we will require formal definitions. But before we supply these, it is instructive to work informally with programs of $\mathscr{S}$.

2. Some Examples of Programs

(a) Our first example is the program

$$[A] \qquad X \leftarrow X - 1$$
$$Y \leftarrow Y + 1$$
$$\text{IF } X \neq 0 \text{ GOTO } A$$

If the initial value x of X is not 0, the effect of this program is to copy x into Y and to decrement the value of X down to 0. (By our conventions the initial value of Y is 0.) If $x = 0$, then the program halts with Y having the value 1. We will say that this program ***computes*** the function

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x & \text{otherwise.} \end{cases}$$

This program halts when it executes the third instruction of the program with X having the value 0. In this case the condition $X \neq 0$ is not fulfilled and therefore the branch is not taken. When an attempt is made to move on to the nonexistent fourth instruction, the program halts. A program will also halt if an instruction labeled L is to be executed, but there is no instruction in the program with that label. In this case, we usually will use the letter E (for "exit") as the label which labels no instruction.

(b) Although the preceding program is a perfectly well-defined program of our language $\mathscr{S}$, we may think of it as having arisen in an attempt to write a program that copies the value of X into Y, and therefore containing a "bug" because it does not handle 0 correctly. The following slightly more complicated example remedies this situation.

```
[A]   IF X ≠ 0 GOTO B
      Z ← Z + 1
      IF Z ≠ 0 GOTO E
[B]   X ← X − 1
      Y ← Y + 1
      Z ← Z + 1
      IF Z ≠ 0 GOTO A
```

As we can easily convince ourselves, this program does copy the value of X into Y for all initial values of X. Thus, we say that it computes the function $f(x) = x$. At first glance. Z's role in the computation may not be obvious. It is used simply to allow us to code an ***unconditional branch.*** That is, the program segment

```
Z ← Z + 1
IFZ ≠ OGOTOL                    (2.1)
```

has the effect (ignoring the effect on the value of Z) of an instruction

GOT0 L

such as is available in most programming languages. To see that this is true we note that the first instruction of the segment guarantees that Z has a nonzero value. Thus the condition $Z \neq 0$ is always true and hence the next instruction performed will be the instruction labeled L. Now GOT0 L is

not an instruction in our language $\mathscr{S}$, but since we will frequently have use for such an instruction, we can use it as an abbreviation for the program segment (2.1). Such an abbreviating pseudoinstruction will be called a ***macro*** and the program or program segment which it abbreviates will be called its ***macro expansion.***

The use of these terms is obviously motivated by similarities with the notion of a macro instruction occurring in many programming languages. At this point we will not discuss how to ensure that the variables local to the macro definition are distinct from the variables used in the main program. Instead, we will manually replace any such duplicate variable uses with unused variables. This will be illustrated in the "expanded" multiplication program in (e). In Section 5 this matter will be dealt with in a formal manner.

(c) Note that although the program of (b) does copy the value of X into Y, in the process the value of X is "destroyed" and the program terminates with X having the value 0. Of course, typically, programmers want to be able to copy the value of one variable into another without the original being "zeroed out." This is accomplished in the next program. (Note that we use our macro instruction GOT0 L several times to shorten the program. Of course, if challenged, we could produce a legal program of $\mathscr{S}$ by replacing each GOT0 L by a macro expansion. These macro expansions would have to use a local variable other than Z so as not to interfere with the value of Z in the main program.)

[A]	If X $\neq$ 0 GOTO B
	GOT0 C
[B]	$X \leftarrow X - 1$
	$Y \leftarrow Y + 1$
	$Z \leftarrow Z + 1$
	GOT0 A
[C]	IF $Z \neq 0$ GOTO D
	GOT0 E
[D]	$Z \leftarrow Z - 1$
	$X \leftarrow X + 1$
	GOT0 C

In the first loop, this program copies the value of X into both Y and Z, while in the second loop, the value of X is restored. When the program terminates, both X and Y contain X's original value and $z = 0$.

We wish to use this program to justify the introduction of a macro which we will write

$$V \leftarrow V'$$

the execution of which will replace the contents of the variable V by the contents of the variable V' while leaving the contents of V' unaltered. Now, this program (c) functions correctly as a copying program only under our assumption that the variables Y and Z are initialized to the value 0. Thus, we can use the program as the basis of a macro expansion of $V \leftarrow V'$ only if we can arrange matters so as to be sure that the corresponding variables have the value 0 whenever the macro expansion is entered. To solve this problem we introduce the macro

$$V \leftarrow 0$$

which will have the effect of setting the contents of V equal to 0. The corresponding macro expansion is simply

$$\begin{array}{ll} [L] & V \leftarrow V - 1 \\ & \text{IF } V \neq 0 \text{ GOTO } L \end{array}$$

where, of course, the label L is to be chosen to be different from any of the labels in the main program. We can now write the macro expansion of $V \leftarrow V'$ by letting the macro $V \leftarrow 0$ precede the program which results when X is replaced by V' and Y is replaced by V in program (c). The result is as follows:

$$\begin{array}{ll} & V \leftarrow 0 \\ [A] & \text{IF } V' \neq 0 \text{ GOTO } B \\ & \text{GOT0 C} \\ [B] & V' \leftarrow V' - 1 \\ & V \leftarrow V + 1 \\ & Z \leftarrow Z + 1 \\ & \text{GOT0 } A \\ [C] & \text{IF } Z \neq 0 \text{ GOTO } D \\ & \text{GOT0 } E \\ [D] & Z \leftarrow Z - 1 \\ & V' \leftarrow V' + 1 \\ & \text{GOT0 C} \end{array}$$

With respect to this macro expansion the following should be noted:

1. It is unnecessary (although of course it would be harmless) to include a $Z \leftarrow 0$ macro at the beginning of the expansion because, as has already been remarked, program (c) terminates with $z = 0$.
2. When inserting the expansion in an actual program, the variable Z will have to be replaced by a local variable which does not occur in the main program.

3. Likewise the labels A, B, C, *D* will have to be replaced by labels which do not occur in the main program.
4. Finally, the label *E* in the macro expansion must be replaced by a label *L* such that the instruction which follows the macro in the main program (if there is one) begins $[L]$.

(d) A program with two inputs that computes the function

$$f(x_1, x_2) = x_1 + x_2$$

is as follows:

```
        Y ← X1
        Z ← X2
[B]     IF Z ≠ 0 GOTO A
        GOT0 E
[A]     Z ← Z − 1
        Y ← Y + 1
        GOT0 B
```

Again, if challenged we would supply macro expansions for "$Y \leftarrow X_1$" and "$Z \leftarrow X_2$" as well as for the two unconditional branches. Note that Z is used to preserve the value of X_2.

(e) We now present a program that multiplies, i.e. that computes $f(x_1, x_2) = x_1 \cdot x_2$. Since multiplication can be regarded as repeated addition, we are led to the "program"

```
        Z2 ← X2
[B]     IF Z2 ≠ 0 GOTO A
        GOT0 E
[A]     Z2 ← Z2 − 1
        Z1 ← X1 + Y
        Y ← Z1
        GOT0 B
```

Of course, the "instruction" $Z_1 \leftarrow X_1$ + Y is not permitted in the language $\mathscr{S}$. What we have in mind is that since we already have an addition program, we can replace the macro $Z_1 \leftarrow X_1$ + Y by a program for computing it, which we will call its macro expansion. At first glance, one might wonder why the pair of instructions

```
Z1 ← X1 + Y
Y ← Z1
```

was used in this program rather than the single instruction

$$Y \leftarrow X_1 + Y$$

since we simply want to replace the current value of Y by the sum of its value and x_1. The sum program in (d) computes Y = $X_1 + X_2$. If we were to use that as a template, we would have to replace X_2 in the program by Y. Now if we tried to use Y also as the variable being assigned, the macro expansion would be as follows:

```
        Y ← X1
        Z ← Y
[B]     IF Z ≠ 0 GOTO A
        GOT0 E
[A]     Z ← Z − 1
        Y ← Y + 1
        GOT0 B
```

What does this program actually compute? It should not be difficult to see that instead of computing x_1 + y as desired, this program computes $2x_1$. Since X_1 is to be added over and over again, it is important that X_1 not be destroyed by the addition program. Here is the multiplication program, showing the macro expansion of $Z_1 \leftarrow X_1 + Y$:

```
        Z2 ← X2
[B]     IF Z2 ≠ 0 GOTO A
        GOT0 E
[A]     Z2 ← Z2 − 1
        Z1 ← X1
        Z3 ← Y
[B2]    IF Z3 ≠ 0 GOTO A2      Macro Expansion of
        GOT0 E2                Z1 ← X1 + Y
[A2]    Z3 ← Z3 − 1
        Z1 ← Z1 + 1
        GOT0 B2
[E2]    Y ← Z1
        GOT0 B
```

Note the following:

1. The local variable Z_1 in the addition program in (d) must be replaced by another local variable (we have used Z_3) because Z_1 (the other name for Z) is also used as a local variable in the multiplication program.

2. The labels A, B, E are used in the multiplication program and hence cannot be used in the macro expansion. We have used A,, B_2, E_2 instead.
3. The instruction GOT0 E_2 terminates the addition. Hence, it is necessary that the instruction immediately following the macro expansion be labeled E_2.

In the future we will often omit such details in connection with macro expansions. All that is important is that our infinite supply of variables and labels guarantees that the needed changes can always be made.

(f) For our final example, we take the program

$$\begin{array}{ll} & Y \leftarrow X_1 \\ & Z \leftarrow X_2 \\ [C] & \text{IF } Z \neq 0 \text{ GOTO } A \\ & \text{GOT0 } E \\ [A] & \text{IF } Y \neq 0 \text{ GOTO } B \\ & \text{GOT0 } A \\ [B] & Y \leftarrow Y - 1 \\ & Z \leftarrow Z - 1 \\ & \text{GOT0 C} \end{array}$$

If we begin with $X_1 = 5$, $X_2 = 2$, the program first sets Y = 5 and $Z = 2$. Successively the program sets Y = 4, $Z = 1$ and Y = 3, $Z = 0$. Thus, the computation terminates with Y = 3 = 5 − 2. Clearly, if we begin with $X_1 = m$, $X_2 = n$, where $m \geq n$, the program will terminate with Y = $m - n$.

What happens if we begin with a value of X_1 less than the value of X_2, e.g., $X_1 = 2$, $X_2 = 5$? The program sets Y = 2 and $Z = 5$ and successively sets Y = 1, $Z = 4$ and Y = 0, $Z = 3$. At this point the computation enters the "loop":

$$\begin{array}{ll} [A] & \text{IF } Y \neq 0 \text{ GOTO } B \\ & \text{GOT0 } A \end{array}$$

Since y = 0, there is no way out of this loop and the computation will continue "forever." Thus, if we begin with $X_1 = m$, $X_2 = n$, where $m < n$, the computation will never terminate. In this case (and in similar cases) we will say that the program computes the *partial function*

$$g(x_1, x_2) = \begin{cases} x_1 - x_2 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2. \end{cases}$$

(Partial functions are discussed in Chapter 1, Section 2.)

Exercises

1. Write a program in $\mathscr{S}$ (using macros freely) that computes the function $f(x) = 3x$.
2. Write a program in $\mathscr{S}$ that solves Exercise 1 using no macros.
3. Let $f(x) = 1$ if x is even; $f(x) = 0$ if x is odd. Write a program in $\mathscr{S}$ that computes f.
4. Let $f(x) = 1$ if x is even; $f(x)$ undefined if x is odd. Write a program in $\mathscr{S}$ that computes f.
5. Let $f(x_1, x_2) = 1$ if $x_1 = x_2$; $f(x_1, x_2) = 0$ if $x_1 \neq x_2$. Without using macros, write a program in $\mathscr{S}$ that computes f.
6. Let f(x) be the greatest number n such that $n^2 \leq x$. Write a program in $\mathscr{S}$ that computes f.
7. Let $\gcd(x_1, x_2)$ be the greatest common divisor of x_1 and x_2. Write a program in $\mathscr{S}$ that computes gcd.

3. Syntax

We are now ready to be mercilessly precise about the language $\mathscr{S}$. Some of the description recapitulates the preceding discussion.

The symbols

$$X_1 \ X_2 \ X_3 \ \cdots$$

are called ***input variables,***

$$Z_1 \ Z_2 \ Z_3 \ \cdots$$

are called ***local variables,*** and Y is called the ***output variable*** of $\mathscr{S}$. The symbols

$$A_1 \ B_1 \ C_1 \ D_1 \ E_1 \ A_2 \ B_2 \ \cdots$$

are called ***labels*** of $\mathscr{S}$. (As already indicated, in practice the subscript 1 is often omitted.) A ***statement*** is one of the following:

$$V \leftarrow V + 1$$
$$V \leftarrow V - 1$$
$$V \leftarrow V$$
$$\text{IF } V \neq 0 \text{ GOTO } L$$

where V may be any variable and L may be any label.

Note that we have included among the statements of $\mathscr{S}$ the "dummy" commands $V \leftarrow V$. Since execution of these commands leaves all values unchanged, they have no effect on what a program computes. They are included for reasons that will not be made clear until much later. But their inclusion is certainly quite harmless.

Next, an ***instruction*** is either a statement (in which case it is also called an *unlabeled* instruction) or $[L]$ followed by a statement (in which case the instruction is said to have L as its label or to be labeled L). A ***program*** is a list (i.e., a finite sequence) of instructions. The length of this list is called the ***length*** of the program. It is useful to include the ***empty program*** of length 0, which of course contains no instructions.

As we have seen informally, in the course of a computation, the variables of a program assume different numerical values. This suggests the following definition:

A ***state of a program*** $\mathscr{P}$ is a list of equations of the form $V = m$, where V is a variable and m is a number, including an equation for each variable that occurs in $\mathscr{P}$ and including no two equations with the same variable. As an example, let $\mathscr{P}$ be the program of (b) from Section 2, which contains the variables X Y Z. The list

$$X = 4, \quad Y = 3, \quad Z = 3$$

is thus a state of $\mathscr{P}$. (The definition of ***state*** does not require that the state can actually be "attained" from some initial state.) The list

$$X_1 = 4, \quad X_2 = 5, \quad Y = 4, \quad Z = 4$$

is also a state of $\mathscr{P}$. (Recall that X is another name for X_1 and note that the definition permits inclusion of equations involving variables not actually occurring in $\mathscr{P}$.) The list

$$X = 3, \quad Z = 3$$

is ***not*** a state of $\mathscr{P}$ since no equation in Y occurs. Likewise, the list

$$X = 3, \quad X = 4, \quad Y = 2, \quad Z = 2$$

is ***not*** a state of $\mathscr{P}$: there are two equations in X.

Let σ be a state of $\mathscr{P}$ and let V be a variable that occurs in σ. The ***value of V at σ*** is then the (unique) number q such that the equation $V = q$ is one of the equations making up σ. For example, the value of X at the state

$$X = 4, \quad Y = 3, \quad Z = 3$$

is 4.

Suppose we have a program $\mathscr{P}$ and a state σ of $\mathscr{P}$. In order to say what happens "next," we also need to know which instruction of $\mathscr{P}$ is about to be executed. We therefore define a ***snapshot*** or ***instantaneous description*** of a program $\mathscr{P}$ of length n to be a pair (i, a) where $1 \leq i \leq n + 1$, and σ is a state of $\mathscr{P}$. (Intuitively the number i indicates that it is the ith instruction which is about to be executed; $i = n + 1$ corresponds to a "stop" instruction.)

If s = (i, a) is a snapshot of $\mathscr{P}$ and V is a variable of $\mathscr{P}$, then the ***value of V at s*** just means the value of V at σ.

A snapshot (i, a) of a program $\mathscr{P}$ of length n is called ***terminal*** if $i = n + 1$. If (i, σ) is a nonterminal snapshot of $\mathscr{P}$, we define the successor of (i, σ) to be the snapshot (j, τ) defined as follows:

Case 1. The ith instruction of $\mathscr{P}$ is $V \leftarrow V + 1$ and σ contains the equation V = m. Then j = $i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m + 1$ (i.e., the value of V at τ is $m + 1$).

Case 2. The ith instruction of $\mathscr{P}$ is $V \leftarrow V - 1$ and σ contains the equation $V = m$. Then j = $i + 1$ and τ is obtained from σ by replacing the equation $V = m$ by $V = m - 1$ if $m \neq 0$; if $m = 0$, $\tau = \sigma$.

Case 3. The ith instruction of $\mathscr{P}$ is $V \leftarrow V$. Then $\tau = \sigma$ and j = $i + 1$.

Case 4. The ith instruction of $\mathscr{P}$ is IF $V \neq 0$ GOT0 L. Then $\tau = \sigma$, and there are two subcases:

Case 4a. σ contains the equation $V = 0$. Then j = $i + 1$.

Case 4b. σ contains the equation $V = m$ where $m \neq 0$. Then, if there is an instruction of $\mathscr{P}$ labeled L, j is the ***least number*** such that the jth instruction of $\mathscr{P}$ is labeled L. Otherwise, j = $n + 1$.

For an example, we return to the program of (b), Section 2. Let σ be the state

$$X = 4, \qquad Y = 0, \qquad Z = 0$$

and let us compute the successor of the snapshots (i, a) for various values of i.

For $i = 1$, the successor is (4, a) where σ is as above. For $i = 2$, the successor is (3, τ), where τ consists of the equations

$$X = 4, \qquad Y = 0, \qquad Z = 1.$$

For $i = 7$, the successor is (8, a). This is a terminal snapshot.

A ***computation*** of a program $\mathscr{P}$ is defined to be a sequence (i.e., a list) $s_1, s_2, \ldots, s_k$ of snapshots of $\mathscr{P}$ such that s_{i+1} is the successor of s_i for $i = 1, 2, \ldots, k - 1$ and s_k is terminal.

Note that we have not forbidden a program to contain more than one instruction having the same label. However, our definition of successor of a snapshot, in effect, interprets a branch instruction as always referring to the first statement in the program having the label in question. Thus, for example, the program

$$\begin{array}{ll} [A] & \text{X + - - X - 1} \\ & \text{IF } X \neq 0 \text{ GOTO } A \\ [A] & X \leftarrow X + 1 \end{array}$$

is equivalent to the program

$$\begin{array}{ll} [A] & X \leftarrow X - 1 \\ & \text{IF } X \neq 0 \text{ GOTO } A \\ & X \leftarrow X + 1 \end{array}$$

Exercises

1. Let $\mathscr{P}$ be the program of(b), Section 2. Write out a computation of $\mathscr{P}$ beginning with the snapshot (1, a), where σ consists of the equations x = 2, Y = 0, $Z = 0$.
2. Give a program $\mathscr{P}$ such that for every computation $s_1, \ldots, s_k$ of $\mathscr{P}$, $k = 5$.
3. Give a program $\mathscr{P}$ such that for any $n \geq 0$ and every computation $s_1 = (1, \sigma), s_2, \ldots, s_k$ of $\mathscr{P}$ that has the equation X = n in σ, $k = 2n + 1$.

4. Computable Functions

We have been speaking of the function computed by a program $\mathscr{P}$. It is now time to make this notion precise.

One would expect a program that computes a function of m variables to contain the input variables $X_1, X_2, \ldots, X_m$, and the output variable Y, and to have all other variables (if any) in the program be local. Although this has been and will continue to be our practice, it is convenient not to make it a formal requirement. According to the definitions we are going to present, any program $\mathscr{P}$ of the language $\mathscr{S}$ can be used to compute a function of one variable, a function of two variables, and, in general, for each $m \geq 1$, a function of m variables.

Thus, let $\mathscr{P}$ be any program in the language $\mathscr{S}$ and let $r_1, \ldots, r_m$ be m given numbers. We form the state σ of $\mathscr{P}$ which consists of the equations

$$X_1 = r_1, \quad X_2 = r_2, \quad \ldots, \quad X_m = r_m, \quad Y = 0$$

together with the equations $V = 0$ for each variable V in $\mathscr{P}$ other than $X_1, \ldots, X_m$, Y. We will call this the ***initial state,*** and the snapshot $(1, \sigma)$, the ***initial snapshot.***

Case 1. There is a computation $s_1, s_2, \ldots, s_k$ of $\mathscr{P}$ beginning with the initial snapshot. Then we write $\psi_{\mathscr{P}}^{(m)}(r_1, r_2, \ldots, r_m)$ for the value of the variable Y at the (terminal) snapshot s_k.

Case 2. There is no such computation; i.e., there is an ***infinite*** sequence $s_1, s_2, s_3, \ldots$ beginning with the initial snapshot where each s_{i+1} is the successor of s_i. In this case $\psi_{\mathscr{P}}^{(m)}(r_1, \ldots, r_m)$ is undefined.

Let us reexamine the examples in Section 2 from the point of view of this definition. We begin with the program of (b). For this program $\mathscr{P}$, we have

$$\psi_{\mathscr{P}}^{(1)}(x) = x$$

for all x. For this one example, we give a detailed treatment. The following list of snapshots is a computation of $\mathscr{P}$:

$$\begin{aligned}
&(1, \{X = r, Y = 0, Z = 0\}),\\
&(4, \{X = r, Y = 0, Z = 0\}),\\
&(5, \{X = r - 1, Y = 0, Z = 0\}),\\
&(6, \{X = r - 1, Y = 1, Z = 0\}),\\
&(7, \{X = r - 1, Y = 1, Z = 1\}),\\
&(1, \{X = r - 1, Y = 1, Z = 1\}),
\end{aligned}$$

$$\begin{aligned}
&(1, \{X = 0, Y = r, Z = r\}),\\
&(2, \{X = 0, Y = r, Z = r\}),\\
&(3, \{X = 0, Y = r, Z = r + 1\}),\\
&(8, \{X = 0, Y = r, Z = r + 1\}).
\end{aligned}$$

We have included a copy of $\mathscr{P}$ showing line numbers:

$[A]$	IF $X \neq 0$ GOTO B	(1)
	$Z \leftarrow Z + 1$	(2)
	IF $Z \neq 0$ GOTO E	(3)
$[B]$	$X \leftarrow X - 1$	(4)
	$Y \leftarrow Y + 1$	(5)
	$Z \leftarrow Z + 1$	(6)
	IF $Z \neq 0$ GOTO A	(7)

For other examples of Section 2 we have

$$\text{(a)} \quad \psi^{(1)}(r) = \begin{cases} 1 & \text{if } r = 0 \\ r & \text{otherwise,} \end{cases}$$

$$\text{(b),(c)} \quad \psi^{(1)}(r) = r,$$

$$\text{(d)} \quad \psi^{(2)}(r_1, r_2) = r_1 + r_2,$$

$$\text{(e)} \quad \psi^{(2)}(r_1, r_2) = r_1 \cdot r_2,$$

$$\text{(f)} \quad \psi^{(2)}(r_1, r_2) = \begin{cases} r_1 - r_2 & \text{if } r_1 \geq r_2 \\ \uparrow & \text{if } r_1 < r_2. \end{cases}$$

Of course in several cases the programs written in Section 2 are abbreviations, and we are assuming that the appropriate macro expansions have been provided.

As indicated, we are permitting each program to be used with any number of inputs. If the program has n input variables, but only $m < n$ are specified, then according to the definition, the remaining input variables are assigned the value 0 and the computation proceeds. If on the other hand, m values are specified where $m > n$ the extra input values are ignored. For example, referring again to the examples from Section 2, we have

$$\text{(c)} \quad \psi_{\mathscr{P}}^{(2)}(r_1, r_2) = r_1,$$

$$\text{(d)} \quad \psi_{\mathscr{P}}^{(1)}(r_1) = r_1 + 0 = r_1,$$

$$\psi_{\mathscr{P}}^{(3)}(r_1, r_2, r_3) = r_1 + r_2.$$

For any program $\mathscr{P}$ and any positive integer m, the function $\psi_{\mathscr{P}}^{(m)}(x_1, \ldots, x_m)$ is said to be ***computed*** by $\mathscr{P}$. A given partial function g (of one or more variables) is said to be ***partially computable*** if it is computed by some program. That is, g is partially computable if there is a program $\mathscr{P}$ such that

$$g(r_1, \ldots, r_m) = \psi_{\mathscr{P}}^{(m)}(r_1, \ldots, r_m)$$

for all $r_1, \ldots, r_m$. Here this 'equation must be understood to mean not only that both sides have the same value when they are defined, but also that when either side of the equation is undefined, the other is also.

As explained in Chapter 1, a given function g of m variables is called ***total*** if $g(r_1, \ldots, r_m)$ is defined for *all* $r_1, \ldots, r_m$. A function is said to be ***computable*** if it is both partially computable and total.

Partially computable functions are also called ***partial recursive,*** and computable functions, i.e., functions that are both total and partial recursive, are called ***recursive. The*** reason for this terminology is largely historical and will be discussed later.

Our examples from Section 2 give us a short list of partially computable functions, namely: $x, x + y$, $x \cdot y$, and $x - y$. Of these, all except the last one are total and hence computable.

Computability theory (also called recursion theory) studies the class of partially computable functions. In order to justify the name, we need some evidence that for every function which one can claim to be "computable" on intuitive grounds, there really is a program of the language $\mathscr{S}$ which computes it. Such evidence will be developed as we go along.

We close this section with one final example of a program of $\mathscr{S}$:

$$\begin{array}{ll} [A] & X \leftarrow X + 1 \\ & \text{IF } X \neq 0 \text{ GOTO } A \end{array}$$

For this program $\mathscr{P}$, $\psi_{\mathscr{P}}^{(1)}(x)$ is undefined for all x. So, the nowhere defined function (see Chapter 1, Section 2) must be included in the class of partially computable functions.

Exercises

1. Let $\mathscr{P}$ be the program

$$\begin{array}{ll} & \text{IF } X \neq 0 \text{ GOTO } A \\ [A] & X \leftarrow X + 1 \\ & \text{IF } X \neq 0 \text{ GOTO } A \\ [A] & Y \leftarrow Y + 1 \end{array}$$

What is $\psi_{\mathscr{P}}^{(1)}(x)$?

2. The same as Exercise 1 for the program

$$\begin{array}{ll} [B] & \text{IF } X \neq 0 \text{ GOTO } A \\ & Z \leftarrow Z + 1 \\ & \text{IF } Z \neq 0 \text{ GOTO } B \\ [A] & X \leftarrow X \end{array}$$

3. The same as Exercise 1 for the empty program.

4. Let $\mathscr{P}$ be the program

$$\begin{array}{ll} & Y \leftarrow X_1 \\ [A] & \text{IF } X_2 = 0 \text{ GOTO } E \\ & Y \leftarrow Y + 1 \\ & Y \leftarrow Y + 1 \\ & X_2 \leftarrow X_2 - 1 \\ & \text{GOTO } A \end{array}$$

What is $\psi_{\mathscr{P}}^{(1)}(r_1)$? $\psi_{\mathscr{P}}^{(2)}(r_1, r_2)$? $\psi_{\mathscr{P}}^{(3)}(r_1, r_2, r_3)$?

5. Show that for every partially computable function $f(x_1, \ldots, x_n)$, there is a number $m \geq 0$ such that f is computed by infinitely many programs of length m.

6. **(a)** For every number $k \geq 0$, let f_k be the constant function $f_k(x) = k$. Show that for every k, f_k is computable.

   (b) Let us call an $\mathscr{S}$ program a ***straightline program*** if it contains no (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOT0 L. Show by induction on the length of programs that if the length of a straightline program $\mathscr{P}$ is k, then $\psi_{\mathscr{P}}^{(1)}(x) \leq k$ for all x.

   (c) Show that, if $\mathscr{P}$ is a straightline program that computes f_k, then the length of $\mathscr{P}$ is at least k.

   (d) Show that no straightline $\mathscr{S}$ program computes the function $f(x) = x + 1$. Conclude that the class of functions computable by straightline $\mathscr{S}$ programs is contained in but is not equal to the class of computable functions.

7. Let us call an $\mathscr{S}$ program $\mathscr{P}$ ***forward-branching*** if the following condition holds for each occurrence in $\mathscr{P}$ of a (labeled or unlabeled) instruction of the form IF $V \neq 0$ GOT0 L. If IF $V \neq 0$ GOT0 L is the ith instruction of $\mathscr{P}$, then either L does not appear as the label of an instruction in $\mathscr{P}$, or else, if j is the least number such that L is the label of the jth instruction in $\mathscr{P}$, then $i < j$. Show that a function is computed by some forward-branching program if and only if it is computed by some straightline program (see Exercise 6).

8. Let us call a unary function ***f(x) partially n-computable*** if it is computed by some $\mathscr{S}$ program $\mathscr{P}$ such that $\mathscr{P}$ has no more than n instructions, every variable in $\mathscr{P}$ is among X, Y, $Z_1, \ldots, Z_n$, and every label in $\mathscr{P}$ is among $A_1, \ldots, A_n, E$.

   (a) Show that if a unary function is computed by a program with no more than n instructions, then it is partially n-computable.

   (b) Show that for every $n \geq 0$, there are only finitely many distinct partially n-computable unary functions.

   (c) Show that for every $n \geq 0$, there are only finitely many distinct unary functions computed by $\mathscr{S}$ programs of length no greater than n.

   (d) Conclude that for every $n \geq 0$, there is a partially computable unary function which is not computed by any $\mathscr{S}$ program of length less than n.

5. More about Macros

In Section 2 we gave some examples of computable functions (i.e., $x + y$, $x \cdot y$) giving rise to corresponding macros. Now we consider this process in general.

Let $f(x_1,\ldots,x_n)$ be some partially computable function computed by the program $\mathscr{P}$. We shall assume that the variables that occur in $\mathscr{P}$ are all included in the list Y, $X_1,\ldots,X_n,Z_1,\ldots,Z_k$ and that the labels that occur in $\mathscr{P}$ are all included in the list $E, A,, \ldots, A,$. We also assume that for each instruction of $\mathscr{P}$ of the form

$$\text{IF } V \neq 0 \text{ GOTO } A_i$$

there is in $\mathscr{P}$ an instruction labeled A_i. (In other words, E is the only "exit" label.) It is obvious that, if $\mathscr{P}$ does not originally meet these conditions, it will after minor changes in notation. We write

$$\mathscr{P}=\mathscr{P}(Y,X_1,\ldots,X_n,Z_1,\ldots,Z_k;E,A,,\ldots,A_l)$$

in order that we can represent programs obtained from $\mathscr{P}$ by replacing the variables and labels by others. In particular, we will write

$$\mathscr{Q}_m=\mathscr{P}(Z_m,Z_{m+1},\ldots,Z_{m+n},Z_{m+n+1},\ldots,Z_{m+n+k};\ E_m,A_{m+1},\ldots,A_{m+l})$$

for each given value of m. Now we want to be able to use macros like

$$W \leftarrow f(V_1,\ldots,V_n)$$

in our programs, where $V_1,\ldots,V_n,W$ can be any variables whatever. (In particular, W might be one of $V_1,\ldots,V_n$.) We will take such a macro to be an abbreviation of the following expansion:

$$\begin{aligned}
&Z_m \leftarrow 0\\
&Z_{m+1} \leftarrow V_1\\
&Z_{m+2} \leftarrow V_2\\
&\vdots\\
&Z_{m+n} \leftarrow V_n\\
&Z_{m+n+1} \leftarrow 0\\
&Z_{m+n+2} \leftarrow 0\\
&\vdots\\
&Z_{m+n+k} \leftarrow 0\\
&\mathscr{Q}_m\\
[E_m]\quad &W \leftarrow Z_m
\end{aligned}$$

Here it is understood that the number m is chosen so large that none of the variables or labels used in $\mathscr{Q}_m$ occur in the main program of which the expansion is a part. Notice that the expansion sets the variables corresponding to the output and local variables of $\mathscr{P}$ equal to 0 and those corresponding to $X_1,\ldots,X_n$ equal to the values of $V_1,\ldots,V_n$, respectively. Setting the variables equal to 0 is necessary (even though they are

all local variables automatically initialized to 0) because the expansion may be part of a loop in the main program; in this case, at the second and subsequent times through the loop the local variables will have whatever values they acquired the previous time around, and so will need to be reset. Note that when $\mathscr{E}_m$ terminates, the value of Z_m is $f(V_1, \ldots, V_n)$, so that W finally does get the value $f(V_1, \ldots, V_n)$.

If $f(V_1, \ldots, V_n)$ is undefined, the program $\mathscr{E}_m$ will never terminate. Thus if f is not total, and the macro

$$W \leftarrow f(V_1, \ldots, V_n)$$

is encountered in a program where $V_1, \ldots, V_n$ have values for which f is not defined, the main program will never terminate.

Here is an example:

$$Z \leftarrow X_1 - X_2$$
$$Y \leftarrow Z + X_3$$

This program computes the function $f(x_1, x_2, x_3)$, where

$$f(x_1, x_2, x_3) = \begin{cases} (x_1 - x_2) + x_3 & \text{if } x_1 \geq x_2 \\ \uparrow & \text{if } x_1 < x_2. \end{cases}$$

In particular, $f(2, 5, 6)$ is undefined, although $(2 - 5) + 6 = 3$ is positive. The computation never gets past the attempt to compute $2 - 5$.

So far we have augmented our language $\mathscr{S}$ to permit the use of macros which allow assignment statements of the form

$$W \leftarrow f(V_1, \ldots, V_n),$$

where f is any partially computable function. Nonetheless there is available only one highly restrictive conditional branch statement, namely,

$$\text{IF } V \neq 0 \text{ GOTO } L$$

We will now see how to augment our language to include macros of the form

$$\text{IF } P(V_1, \ldots, V_n) \text{ GOT0 } L$$

where $P(x_1, \ldots, x_n)$ is a computable predicate. Here we are making use of the convention, introduced in Chapter 1, that

$$\text{TRUE} = 1, \qquad \text{FALSE} = 0.$$

Hence predicates are just total functions whose values are always either 0 or 1. And therefore, it makes perfect sense to say that some given ***predicate*** is or is not computable.

Let $P(x_1,\ldots,x_n)$ be any computable predicate. Then the appropriate macro expansion of

$$\text{IF } P(V_1,\ldots,V_n) \text{ GOT0 } L$$

is simply

$$Z \leftarrow P(V_1,\ldots,V_n)$$
$$\text{IF } Z \neq 0 \text{ GOTO } L$$

Note that P is a computable function and hence we have already shown how to expand the first instruction. The second instruction, being one of the basic instructions in the language $\mathscr{S}$, needs no further expansion.

A simple example of this general kind of conditional branch statement which we will use frequently is

$$\text{IF } V = 0 \text{ GOTO } L$$

To see that this is legitimate we need only check that the-predicate $P(x)$, defined by $P(x)$ = TRUE if $x = 0$ and $P(x)$ = FALSE otherwise, is computable. Since TRUE = 1 and FALSE = 0, the following program does the job:

$$\text{IF } X \neq 0 \text{ GOTO } E$$
$$Y \leftarrow Y + 1$$

The use of macros has the effect of enabling us to write much shorter programs than would be possible restricting ourselves to instructions of the original language $\mathscr{S}$. The original "assignment" statements $V \leftarrow V + 1$, $V \leftarrow V - 1$ are now augmented by general assignment statements of the form $W \leftarrow f(V_1,\ldots,V_n)$ for any partially computable function f. Also, the original conditional branch statements IF $V \neq 0$ GOT0 L are now augmented by general conditional branch statements of the form IF $P(V_1,\ldots,V_n)$ GOT0 L for any computable predicate P. The fact that any function which can be computed using these general instructions could already have been computed by a program of our original language $\mathscr{S}$ (since the general instructions are merely abbreviations of programs of $\mathscr{S}$) is powerful evidence of the generality of our notion of computability.

Our next task will be to develop techniques that will make it easy to see that various particular functions are computable.

Exercises

1. (a) Use the process described in this section to expand the program in example (d) of Section 2.
   (b) What is the length of the $\mathscr{S}$ program expanded from example (e) by this process?
2. Replace the instructions

$$Z_1 \leftarrow X_1 + Y$$
$$Y \leftarrow Z_1$$

   in example (e) of Section 2 with the instruction $Y \leftarrow X_1 + Y$, and expand the result by the process described in this section. If $\mathscr{P}$ is the resulting $\mathscr{S}$ program, what is $\psi_{\mathscr{P}}^{(2)}(r_1, r_2)$?
3. Let $f(x), g(x)$ be computable functions and let h(x) = $f(g(x))$. Show that h is computable.
4. Show by constructing a program that the predicate $x_1 \leq x_2$ is computable.
5. Let $P(x)$ be a computable predicate. Show that the function f defined by

$$f(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{if } P(x_1 + x_2) \\ \uparrow & \text{otherwise} \end{cases}$$

   is partially computable.
6. Let $P(x)$ be a computable predicate. Show that

$$EX_P(r) = \begin{cases} 1 & \text{if there are at least } r \text{ numbers } n \text{ such that } P(n) = 1 \\ \uparrow & \text{otherwise} \end{cases}$$

   is partially computable.
7. Let π be a computable permutation (i.e., one-one, onto function) of N, and let π^{-1} be the inverse of π, i.e.,

$$\pi^{-1}(y) = x \quad \text{if and only if} \quad \pi(x) = y.$$

   Show that π^{-1} is computable.
8. Let $f(x)$ be a partially computable but not total function, let M be a finite set of numbers such that $f(m)\uparrow$ for all $m \in M$, and let g(x) be

an arbitrary partially computable function. Show that

$$h(x) = \begin{cases} g(x) & \text{if } x \in M \\ f(x) & \text{otherwise} \end{cases}$$

is partially computable.

9. Let $\mathscr{S}^+$ be a programming language that extends $\mathscr{S}$ by permitting instructions of the form $V \leftarrow k$, for any $k \geq 0$. These instructions have the obvious effect of setting the value of V to k. Show that a function is partially computable by some $\mathscr{S}^+$ program if and only if it is partially computable.

10. Let $\mathscr{S}'$ be a programming language defined like $\mathscr{S}$ except that its (labeled and unlabeled) instructions are of the three types

$$V \leftarrow V'$$
$$V \leftarrow V + 1$$
$$\text{If } V \neq V' \text{ GOTO } L$$

These instructions are given the obvious meaning. Show that a function is partially computable in $\mathscr{S}'$ if and only if it is partially computable.