Technical Report CS75030-T

COMPUTABILITY THEORY
An Introduction for Students of Computer Science

by

Andy N.C. Kang

Department of Computer Science

College of Arts and Sciences
Virginia Polytechnic Institute & State University
Blacksburg, Virginia 24061

January 1976

This work is subject to copyright. All rights are reserved.
© by Andy N.C. Kang, Department of Computer Science, V.P.I. & S.U.

CONTENTS

PREFACE

The primary goal of this book is to introduce the basic concepts of effective computability and to prepare the reader for the study of formal language theory, recursive function theory, and the theory of computational complexity. As opposed to the conventional treatments on effective computability which are oriented towards the foundations of mathematics, the present text relates the subject matter to computer science. This book intends to present computability theory at a level suitable to junior or senior computer science students. The main prerequisite is the ability to follow mathematical proofs. Therefore, no formal background is needed except for a general acquaintance with mathematical formalism. Programming knowledge obtained in the first two years of computer science curriculum is helpful.

This book consists of six chapters. In the first three chapters some basic results in computability theory are established through the study of a simple computer model. Chapter 1 introduces this computer model, the program machine. In Chapter 2, the class of computable functions defined by a subset of program machines is identified to be the class of primitive recursive functions. Based on the concept of a stored-program computer, a universal program machine is constructed in Chapter 3. As a byproduct, the equivalence of program computable functions and partial recursive functions is established. In Chapter 4, the characterization of effective computability via Turing machines is introduced. A discussion is also included there on an informal approach to verify, by Church's Thesis, recursiveness without going through tedious details of constructing formal notions of effective computability. In Chapter 5, some unsolvable problems are presented which

serve to point out the limitations of any effective procedure. In Chapter 6, recursive and recursively enumerable sets are studied. Some properties of these sets are directly applicable to the theory of formal languages.

This book can be used as a textbook on computability theory at an introductory level. With some programming experience, junior or senior students can absorb most of the material in a one-quarter term. Exercises for each chapter are grouped together in the last section of that chapter.

Chapter 1

THE PROGRAM MACHINE

1.1 Program Machines and Programs

One of the major objectives of this book is to introduce the concept of effective procedure and to develop the reader an understanding of the class of computable functions. Motivated by the concept of computer programming, we shall formalize the notion of effective procedure through a simple model of a digital computer, the program machine. This model is simple enough to be treated mathematically, yet it preserves most features that a digital computer has.

The major difference between a program machine and a digital computer is on their memory structures. A digital computer has a large number of addressable finite-sized memory units. Rather, a program machine has only a small number of addressable infinite-sized memory units, called registers. Infinite-sized register means that each register is big enough to hold any finite nonnegative integer whatsoever. In this book, we denote these registers by small letters (possibly with subscripts) such as x_1, x_2, y_1, z, etc. The program machine also provides a processing unit, which can inspect the contents of any register and perform certain operations on these registers. The operations carried out by a program machine are determined by a program, which is a finite sequence of instructions built in the program machine. A number is associated with each instruction in this sequence. In general, each instruction specifies (1) an operation, (2) one or two registers and (3) the instruction number of one or two other instructions. We now discuss the initial set of four instructions:

(a) Zero: $x_i \leftarrow 0$. Set the contents of register x_i to zero. Go on to the next instruction in the sequence.

(b) Successor: $x_i \leftarrow x_i+1$. Add 1 to the contents of register x_i. Go on to the next instruction in the sequence.

(c) Jump on not equal: *if* $x_i \neq x_j$ *then* n. If the contents of x_i and x_j differ, jump to the nth instruction in the sequence. If they agree, go on to the next instruction in the sequence. The contents of x_i and x_j remain unchanged.

(d) Exit: *exit* Stop the machine.

If an instruction "*if* $x_i \neq x_j$ *then* n" occurs in a program, n must be less than the total number of instructions of the program. In other words, the next instruction referred to in the current instruction must appear somewhere in the sequence. Note that a program machine is different from a stored-program computer for the program machine is capable of performing only one job specified by its built-in program. Since the functions of a program machine are completely specified by its program, we shall use the term "program machine" and "program" interchangeably. Now we describe a simple program machine in Fig. 1. The instruction number and the instruction are separated by a colon ":".

Instruction number	Instruction
1:	$x_3 \leftarrow 0$
2:	$x_4 \leftarrow 0$
3:	$x_4 \leftarrow x_4+1$
4:	$x_1 \leftarrow 0$
5:	*if* $x_2 \neq x_1$ *then* 7
6:	*exit*
7:	$x_1 \leftarrow x_1+1$
8:	*if* $x_3 \neq x_4$ *then* 5

Fig. 1.

The machine begins with instruction number 1. If x_2 initially contains a number a_2, the machine will eventually stop with the same number a_2 in x_1. To see this, we trace the operation, starting with 2 in x_2 and any numbers in x_1, x_3, x_4:

Instruction number	Effect	Register contents			
		x_1	x_2	x_3	x_4
1	Set x_3 to 0	-	2	0	-
2	Set x_4 to 0	-	2	0	0
3	Increase x_4 by 1	-	2	0	1
4	Set x_1 to 0	0	2	0	1
5	Contents of x_1 and x_1 differ so go to 7	0	2	0	1
7	increase x_1 by 1	1	2	0	1
8	Contents of x_3 and x_4 differ so go to 5	1	2	0	1
5	Contents of x_2 and x_1 differ so go to 7	1	2	0	1
7	increase x_1 by 1	2	2	0	1
8	Contents of x_3 and x_4 differ so go to 5	2	2	0	1
5	Contents of x_2 and x_1 agree so go to 6	2	2	0	1
6	halt				

We are using the registers x_3 and x_4 in a wasteful manner - they serve only as a device for getting the program to go back to an earlier instruction. We will use a new instruction "goto n" for this purpose, with the understanding that it can be replaced by using only the original set of instructions on two extra registers. With this new instruction, the above program looks like:

Instruction number	Instruction
1:	$x_1 \leftarrow 0$
2:	if $x_2 \neq x_1$ then 4
3:	exit
4:	$x_1 \leftarrow x_1+1$
5:	goto 2

Fig. 2.

This program essentially copies the contents of x_2 into x_1 and leaves the contents of x_2 unchanged. It is convenient to define a new instruction of this type. Let us denote this instruction by "$x_1 \leftarrow x_2$". Again, we understand that the "copy" instruction can always be implemented by using only the instructions belonging to the original set.

Let us look at another example which performs subtraction. Since we deal with the set of nonnegative integers, the instruction "$x_i \leftarrow x_i \dot{-} 1$" is defined as to decrease the contents of x_i by one only when x_i contains a positive number; in case x_i contained 0, its content remains to be 0.

The following program machine has three registers x_1, x_2, x_3. The program is shown in Fig. 3 below.

Instruction number	Instruction
1:	$x_1 \leftarrow 0$
2:	$x_3 \leftarrow 0$
3:	$x_3 \leftarrow x_3+1$
4:	if $x_2 \neq x_1$ then 8
5:	exit
6:	$x_1 \leftarrow x_1+1$
7:	$x_3 \leftarrow x_3+1$
8:	if $x_2 \neq x_3$ then 6
9:	exit

Fig. 3.

The machine begins with instruction number 1 and if x_2 initially contains 0, the machine will halt at instruction number 5 and x_1 will also contain 0. On the other hand, if x_2 initially contains $a_2 \neq 0$, the machine will eventually halt at instruction number 9 and x_1 will contain $a_2 - 1$. That is, this program performs the instruction $x_1 \leftarrow x_2 \dot{-} 1$.

It is usually convenient to represent a program by a flowchart. An example should suffice to illustrate the procedure to pass from programs to flowcharts, and vice versa. The flowchart for the program in Fig. 3 is shown in Fig. 4.

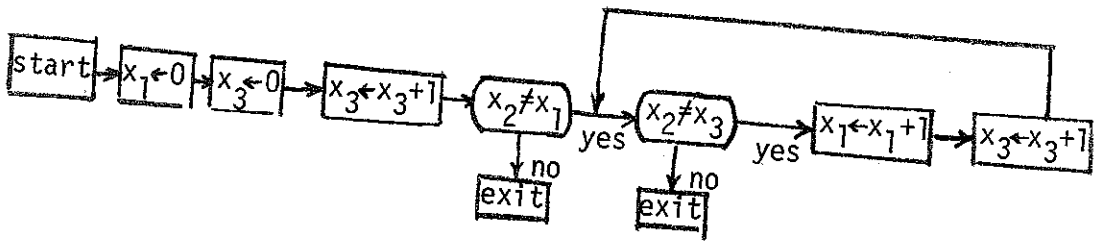


Figure 4

From now on, we shall use flowchart and list representation of programs interchangeably.

1.2 Functions defined by program machines

While it may be intuitively clear what the computation is carried out by a program machine, we still need a formal definition. The easiest way to understand the computation performed by a program machine is to consider the actions of the instruction as being taken at discrete step 1, 2, 3, ... etc. At step i the contents of the registers $x_1, \ldots, x_k$ specified in the program machine are denoted by a k-tuple $\underline{a}_i = \langle a_{i1}, \ldots, a_{ik} \rangle$. For instance, if the instruction to be executed at step i, say, is "$x_j \leftarrow x_j+1$", and its associated instruction number is λ_i, then at step i+1 the contents of these k registers become $\underline{a}_{i+1} = \langle a_{i1}, a_{i2}, \ldots, a_{ij+1}, \ldots, a_{ik} \rangle$, and the instruction to be executed at the next step will be the one associated with the instruction number $\lambda_i + 1$, i.e., $\lambda_{i+1} = \lambda_i + 1$. In general, the pair $\langle \lambda_i, \underline{a}_i \rangle$, consisting of the instruction number λ_i of the instruction to be executed and the k-tuple $\underline{a}_i$ of the contents of the k registers before execution, determines the next pair $\langle \lambda_{i+1}, a_{i+1} \rangle$. The entire record of the computation is therefore made explicit if we give the sequence of these pairs $\langle \lambda_1, \underline{a}_1 \rangle, \langle \lambda_2, \underline{a}_2 \rangle, \ldots$ This concept is used to identify the function computed by a program machine.

Definition

Let P be a program in which only $x_1, x_2, \ldots, x_k$ occur. A computation

according to P is a finite or infinitive sequence of pairs $< \lambda_1, \underline{a}_1 >$, $< \lambda_2, \underline{a}_2 >$, ... such that

(1) λ_1 is 1 and $\underline{a}_1 = < a_{11} a_{12}, \cdots, a_{1k} >$ is the contents of the registers $x_1, x_2, \cdots x_k$ initially.

(2) λ_{i+1} is the instruction number and $\underline{a}_{i+1}$ is the k-tuple of the contents of $x_1, x_2, \cdots, x_k$ obtained after the instruction with number λ_i is performed on the k-tuple of $\underline{a}_i$.

Definition

A computation terminates if it contains a pair $< \lambda_s, \underline{a}_s >$ where the instruction with number λ_s is an exit instruction in P. In this case, we call $\underline{a}_s$ the output of the computation for the input $\underline{a}_1$.

Under the above notion, a program P defines a partial function Φ from N^k into N^k as follows: Given a k-tuple $\underline{a}_1$, if the computation according to P terminates at a finite step, s, the function value is then defined as $\underline{a}_s$. On the other hand, if the computation does not terminate, the function value is then said to be undefined. The partial function Φ is said to be the function defined by P.

As an example, a computation according to the program in Fig. 3 is,

< 1, < 5, 3, 2>>	input $x_1 = 5$, $x_2 = 3$, $x_3 = 2$
< 2, < 0, 3, 2>>	instruction 1 is executed
< 3, < 0, 3, 0>>	instruction 2 is executed
< 4, < 0, 3, 1>>	instruction 3 is executed
< 8, < 0, 3, 1>>	x_1 and x_2 differ
< 6, < 0, 3, 1>>	x_2 and x_3 differ
< 7 < 1, 3, 1>>	instruction 6 is executed
< 8, < 1, 3, 2>>	instruction 7 is executed
< 6, < 1, 3, 2>>	x_2 and x_3 differ
< 7, < 2, 3, 2>>	instruction 6 is executed
< 8, < 2, 3, 3>>	instruction 7 is executed
< 9, < 2, 3, 3>>	x_2 and x_3 agree

We found instruction number 9 is an exit instruction, the computation therefore terminates. The output is $<2,3,3>$ for the input $<5,3,2>$. We consider now an input to this program, say, $<a_1,a_2,a_3>$. After termination of the computation, the registers shall have final output, say, $<b_1,b_2,b_3>$. The relation between input and output can be represented by single function Φ: $<b_1, b_2, b_3> = \Phi(<a_1,a_2,a_3>)$, or can be represented by three functions ϕ_1,ϕ_2,ϕ_3, one for each register: $b_1 = \phi_1 (a_1,a_2, a_3)$, $b_2 = \phi_2 (a_1,a_2, a_3)$, and $b_3 = \phi_3(a_1,a_2,a_3)$. In the present example, ϕ_1,ϕ_2, and ϕ_3 are total functions. In fact, the functions defined by this program are:

$$\phi_1 (a_1, a_2,a_3) = a_2 \dot{-} 1$$

$$\phi_2 (a_1, a_2,a_3) = a_2$$

$$\phi_3 (a_1,a_2,a_3) = \begin{cases} a_2 & \text{if } a_2 \neq 0, \\ 1 & \text{if } a_2 = 0. \end{cases}$$

1.3 Program-computable functions

The registers that occur in a program play quite different roles. Most often we are only interested in knowing the contents of one of the registers. This register is usually called the output register. In the example of Section 1.2, x_1 is the output register. Another type of registers whose initial value really influences the final outcome of the computation, such as x_2 in that example, are called the input registers. The third type of registers whose initial value does not matter for the output but are used for intermediate storage are called auxiliary registers. x_3 is such a register in that example.

We shall fix our attention on functions that can be defined by program machines. The definition below includes partial functions, that is, functions which are not defined for all arguments.

Definition

A partial function ϕ of m variables is said to be program computable (in short, computable) if there exists a program P with m input registers $x_1, x_2, \cdots, x_m$, n auxiliary registers $y_1, y_2, \cdots, y_n$ $(n \geq 0)$, and an output register z such that the following conditions are satisfied:

(1) if $\phi(a_1, \cdots a_m)$ is defined, then the computation according to P terminates on input $< a_1, \ldots, a_m, 0, \ldots 0, 0>$ with output $< b_1, \ldots, b_m, d_1, \ldots, d_n, c>$ where $c = \phi(a_1, \ldots, a_m)$.

(2) if $\phi(a_1, \ldots, a_m)$ is undefined then the computation according to P on input $< a_1, \ldots, a_m, 0, \ldots, 0, 0>$ does not terminate.

In case a computable function is defined on all arguments, we say it is total computable. Note that the output register z could be any one of the input registers. In this case, the input and output of the computation according to P are both (m+n)-tuples rather than (m+n+1)-tuples.

As an example, consider the square root function defined below:

$$\phi(x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is a square,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

ϕ is computable since the program shown in Figure 5 computes ϕ. In this program, x is an input register, y is an auxiliary register, and z is the output register. An extensive set of instructions (e.g. $y \leftarrow x \cdot x$, goto 2) is used in this example. Nevertheless, these instructions can always be replaced by a sequence of instructions from the original instruction set with some extra auxiliary registers.

Instruction number	Instruction
1:	$z \leftarrow 0$
2:	$y \leftarrow z \cdot z$
3:	if $y \neq x$ then 5
4:	exit
5:	$z \leftarrow z + 1$
6:	goto 2

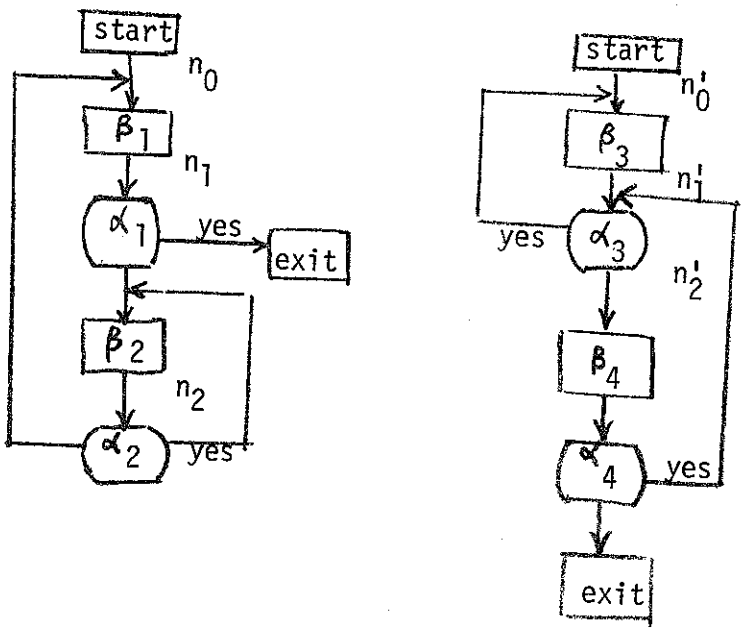
Figure 5

If x is a square, this program will eventually terminate at instruction number 4 and yield an output in register z such that $z \cdot z = x$. That is, $\phi(x) = \sqrt{x}$. On the other hand, if x is not a square, then this program will never terminate. In this case $\phi(x)$ is undefined.

Take another example, in which a partial function ψ is defined in terms of two other partial functions ϕ and ϕ' with disjoint domain i.e., domain $\phi \cap$ domain ϕ' is empty.

$$\psi(x_1,x_2,\ldots x_m) = \begin{cases} \phi(x_1,x_2,\ldots,x_m) & \text{if } \phi(x_1,x_2,\ldots x_m) \text{ is defined,} \\ \phi'(x_1,x_2,\ldots x_m) & \text{if } \phi'(x_1,x_2,\ldots x_m) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If ϕ and ϕ' are computable and are computed by programs P and P' respectively, then ψ is also computable. To see this, we shall design a program which computes both ϕ and ϕ' on a timesharing basis until one halts. This can be achieved by combining the programs P and P' as a coroutine. To illustrate this idea, let us combine the following two programs (shown in Figure 6 as in flowchart form) together as a coroutine.



P P'

Figure 6

In these flowcharts, α's and β's are irrelevant and they can be arbitrary. n_0, n_1, n_2, n'_0, n'_1, n'_2 are the instruction numbers associated with the corresponding boxes in these flowcharts. The combining flowchart is shown in Figure 7 in which two variables w and w' not appearing in P and P' are provided to "remember" the next instruction to be executed when the computation is resumed. Notice that every loop of P and P' is broken to route the control to the other program so that both P and P' get executed on a timesharing basis. Since P and P' are domain disjoint, if one terminates, the output produced is then the function value for ψ.

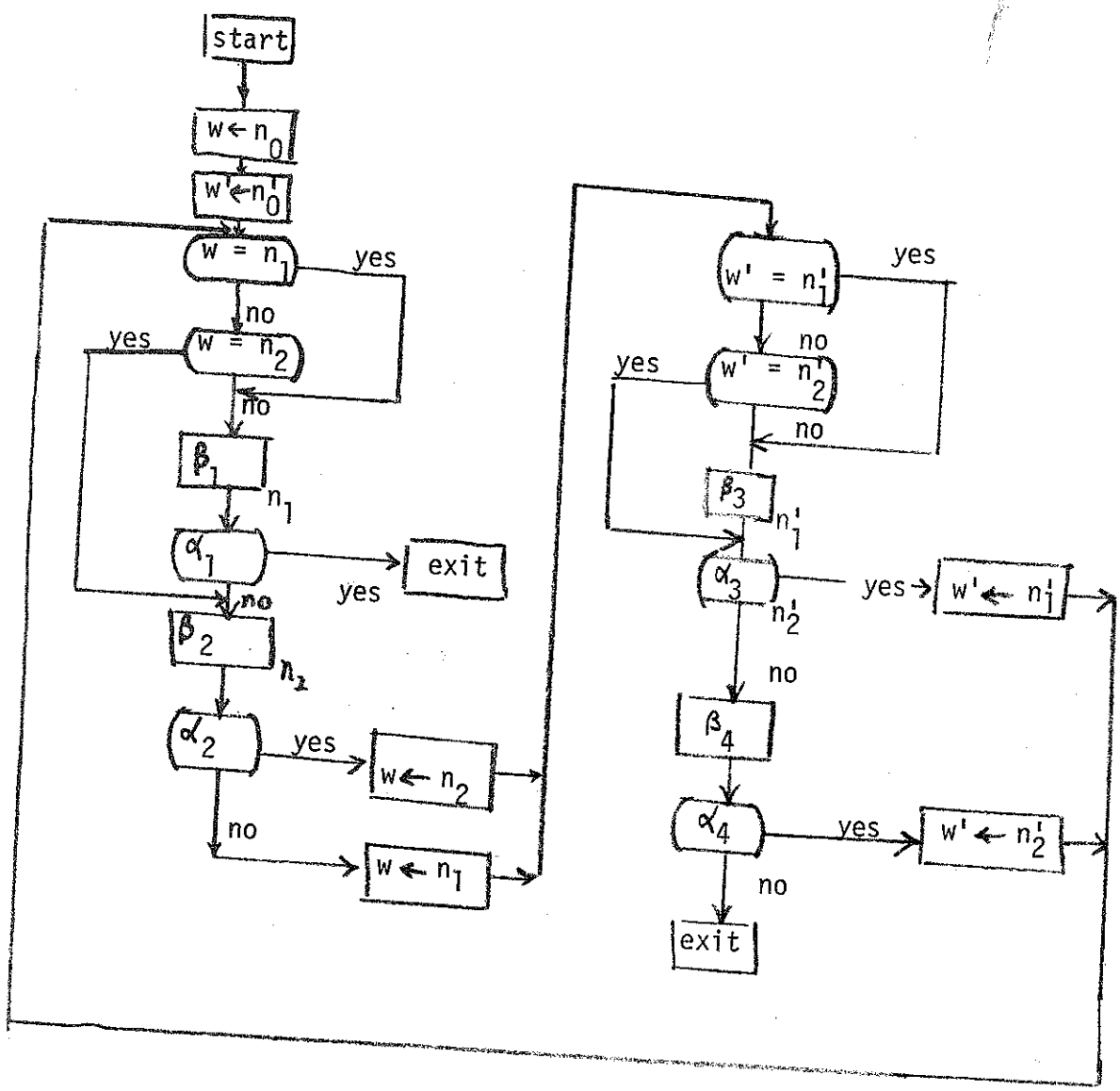

start
w ← n0
w' ← n'0
w = n1
yes
no
yes
w = n2
no
β1
n1
α1
exit
yes
no
β2
n2
α2
yes
w ← n2
no
w ← n1
w' = n'1
yes
no
yes
w' = n'2
no
β3
n'1
α3
yes
w' ← n'1
n'2
no
β4
α4
yes
w' ← n'2
no
exit

Figure 7

Exercise

1. This problem relates another basic set of instructions for program machines.

   a. Consider a new instruction "decrement or jump":

   $$\underline{\text{if}}\ x_i = 0\ \underline{\text{then}}\ n\ \underline{\text{else}}\ x_i \leftarrow x_i - 1,$$

   meaning that if content of x_i is zero, jump to the nth instruction, otherwise decrease it by 1 and go to the next instruction. Show that "Jump on not equal" instruction can be replaced by this instruction together with "zero" and "successor" instructions.

   b. Consider a new instruction "decrement and jump":

   $$\underline{\text{if}}\ x_i \neq 0\ \underline{\text{then}}\ x_i \leftarrow x_i - 1\ \underline{\text{and}}\ n$$

   meaning that if content of x_i is nonzero, decrease it by 1 and go to nth instruction, otherwise go to the next instruction. Show that both "zero" and "jump on not equal" instructions can be replaced by this instruction together with the "successor" instruction.

2. Show the following functions are computable:

   a. $f(0) = 0,\ f(x) = x^{x^{\cdot^{\cdot^{\cdot^{x}}}}}\Big\}\, x$, e.g. $f(3) = 3^{3^3} = 3^{27}$.

   b. $f(x,0) = x,\ f(x,y) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{x}}}}}}\Big\}\, y$, e.g. $f(x,2) = 2^{(2^x)}$.

   c. $f(0) = 2,\ f(1) = 3,\ f(x) = (x+1)$st prime number.

3. Define a function recursively in terms of a total function g.

   $$f(x,0) = g(x), \quad f(x,y+1) = f(f(x,y),y).$$

   If g is computable, show f is also computable.

4. Let $\{\phi_i(x_1,\ldots,x_m) \mid i = 1,\ldots,n\}$ be a set of partial functions with mutually disjoint domain, i.e. for each m-tuple $(x_1,\ldots,x_m)$, at most one of $\phi_i(x_1,\ldots,x_m)$ is defined.

   Define a partial function ψ as follows

$$\psi(x_1,\ldots,x_m) = \begin{cases} \phi_1(x_1,\ldots,x_m) & \text{if } \phi_1(x_1,\ldots,x_m) \text{ is defined,} \\ \vdots & \\ \phi_n(x_1,\ldots,x_m) & \text{if } \phi_n(x_1,\ldots,x_m) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

If $\{\phi_i(x_1,\ldots,x_m),\ i = 1,\ldots,n\}$ are computable, show ψ is also computable.

Reference

Program machine adopted in this chapter was introduced in Minsky [1967]. The idea of using models similar to digital computers to study the class of computable functions was also introduced by Shephardson and Sturgis [1963] and by Wang [1957].

Chapter 2

PRIMITIVE RECURSIVE FUNCTIONS AND LOOP PROGRAMS

One question we would like to ask is what are the functions that can be computed by program machines. In other words, as far as computing functions is concerned what is the capability of a computer. We answer this question for a limited class of programs, the loop programs, in this chapter. The question in general will be answered in subsequent chapters. The functions computed by loop programs are characterized by a class of functions called primitive recursive functions which we are going to define next.

2.1 Primitive Recursive Functions

In this section, we define a class of functions by certain kinds of recursive definition. A recursive definition for a function is a definition for which the values of the function for given arguments are somehow related to values of the same function for "simpler" arguments or to values of other "simpler" functions. For instance, the function defined by

$$f(0)=1,$$
$$f(x+1)=(x+1)\cdot f(x),$$

gives the factorial function.

The primitive recursive functions are an example of a class of functions that can be obtained by such a characterization.

Definition

(1) The following functions are called initial functions.

(a) The zero function: $Z(x)=0$ for all x,

(b) The successor function: $S(x)=x+1$ for all x,

(c) The identity function: $U_i^m(x_1,\ldots,x_m)=x_i$ for all x_1 for all $x_1,\ldots,x_m$.

(2) The following are rules for obtaining new functions from given functions.

(d) Composition

$$f(x_1,\ldots,x_m)=g(h_1(x_1,\ldots,x_m),\ldots,h_n(x_1,\ldots,x_m))$$

f is said to be obtained by composition from the functions

$$g(y_1,\ldots,y_n),\ h(x_1,\ldots,x_m),\ldots,h_n(x_1,\ldots,x_m).$$

(e) Recursion

$$f(x_1,\ldots,x_m,0) = g(x_1,\ldots,x_m),$$

$$f(x_1,\ldots,x_m,y+1) = h(x_1,\ldots,x_m,y,f(x_1,\ldots,x_m,y))$$

f is said to be obtained by recursion from g and h.

Here, we allow m=0, in which case we have

$f(0) = k$ (where k is a fixed non-negative integer)

$f(y+1) = h(y,f(y))$.

Definition

The class of primitive recursive functions is the smallest class of functions that contains the initial functions and is closed under composition and recursion.

That is, a function f is primitive recursive if and only if there is a finite sequence of functions, $f_0,f_1,\ldots,f_k$ such that $f_k=f$, and, for $0 \leq i \leq k$, either f_i is an initial function, or f_i comes from preceding functions in the sequence by an application of composition or recursion.

We now collect a list of useful functions and show that they are primitive recursive.

(1) Addition function:

$x+0 = x$
$x+(y+1) = S(x+y)$

The definition does not fall in with the definition of a primitive recursive function. To be precise, the definition should be

$$f_+(x,0) = x+0 = U_1^1(x)$$

$$f_+(x,y+1) = x+(y+1) = h\ (x,y,f_+(x,y)),$$

where $h(x,y,z) = S(U_3^3(x,y,z))$ is defined by composition.

In the sequel, to identify a primitive recursive function we shall not present its definition in such a formal way, but rather, we only indicate the type of definition employed to define the function in terms of previous known primitive recursive functions.

(2) Multiplication, by recursion in terms of addition function

$$x\cdot 0 = 0$$

$$x\cdot(y+1) = x\cdot y+x$$

(3) Power, by recursion in terms of multiplication function.

$$x^0 = 1$$

$$x^{(y+1)} = x^y \cdot x$$

(4) Predecessor, i.e.

$$P(x) = x\dot{-}1 = \begin{cases} x-1 & \text{if } x\geq 1, \\ 0 & \text{Otherwise,} \end{cases}$$

by recursion in terms of identity function:

$$P(0) = 0$$
$$P(y+1) = y$$

(5) Modified difference, i.e.

$$x\dot{-}y = \begin{cases} x-y & \text{if } x\geq y, \\ 0 & \text{Otherwise,} \end{cases}$$

by recursion in terms of predessor function:

$$x\dot{-}0 = x$$

$$x\dot{-}(y+1) = P(x\dot{-}y)$$

(6) Distance, by composition:

$$|x-y| = (x\dot{-}y) + (y\dot{-}x)$$

The following functions will be technically useful.

(7) $sg(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x \neq 0 \end{cases}$

by recursion: $sg(0) = 0$
$sg(y+1) = 1$

(8) $rm(x,y)$, remainder upon division of y by x,

by recursion: $rm(x,0) = 0$
$rm(x, y+1) = S(rm(x,y)) \dot{-} sg(|x \dot{-} S(rm(x,y))|)$

(9) Divisor function, i.e.

$$D(x,y) = \begin{cases} 1 & \text{if } x \text{ is a divisor of } y, \\ 0 & \text{otherwise,} \end{cases}$$

by composition: $D(x,y) = 1 \dot{-} sg(rm(x,y))$

(10) Bounded summation on primitive recursive function f, i.e.

$$g(x,z) = \sum_{i=0}^{z-1} f(i,x) = \begin{cases} 0 & \text{if } z = 0, \\ f(0,x) + f(1,x) + \ldots + f(z-1,x) & \text{if } z > 0, \end{cases}$$

by recursion: $g(x,0) = 0$
$g(x,z+1) = f(x,z) + g(x,z)$

(11) Bounded multiplication on primitive recursive function f, i.e.

$$g(x,z) = \prod_{i=0}^{z-1} f(i,x) = \begin{cases} 1 & \text{if } z = 0, \\ f(0,x) \cdot f(1,x) \ldots f(z-1,x) & \text{if } z > 0, \end{cases}$$

by recursion: $g(x,0) = 1$
$g(x,z+1) = f(x,z) \cdot g(x,z)$

Notice that a function is not necessarily primitive recursive even if it can be defined by some recurrence equation. For example, the Ackermann's function:

$$a(0,y) = y+1$$
$$a(x+1,0) = a(x,1)$$
$$a(x+1,y+1) = a(x,a(x+1,y)),$$

is defined recursively but it is not primitive recursive. This function does give a unique value for each pair of arguments x and y and appears to be intuitively computable. This provides an intuitive evidence that the class of primitive recursive functions does not include all computable functions.

We shall now show that primitive recursive functions are program computable. The initial functions are: for the zero function is computed by the "zero" instruction; the successor function is computed by the "successor" instruction; and the identity function is computed by the "copy" instruction. In general, To show a function is computable, we need to obtain a program that computes it. However, it is not necessary that the program be completely given. In fact, if a function is defined in terms of functions that have been recognized by a set of programs as computable, all we need then is to obtain a new program by a proper combination of the set of programs following the definition of the function in question.

For example, consider a function f defined by the composition from $g, h_1, \ldots h_n$:

$$f(x_1,\ldots,x_m) = g(h_1(x,\ldots,x_m),\ldots,h_n(x_1,\ldots x_m)),$$

Let us assume $g, h_1,\ldots,h_n$ are computable by a set of programs (represented by hexagonal boxes in Fig. 1 below). Furthermore, we may assume for each of these programs the contents of the input registers remain unchanged after the program terminates. Then the following program obviously computes the function f (z is the output register):

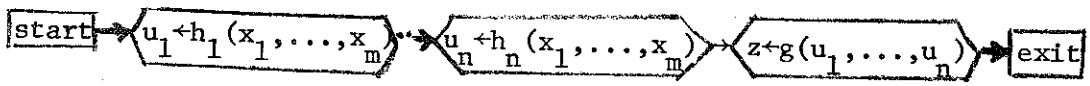


Fig. 1.

As another example, we consider a function f defined in terms of recursion from g and h:

$$f(x_1,\ldots,x_m,0) = g(x_1,\ldots,x_m),$$

$$f(x_1,\ldots,x_m,y+1) = h(x_1,\ldots,x_m,y,f(x_1,\ldots,x_m,y)),$$

Again, we assume g and h are known computable functions and their programs leave the contents of their input registers unchanged after termination. To compute $f(x_1,\ldots,x_m,y)$ for $y > 0$, we need first run the program g on

input $(x_1,\ldots,x_m)$ and obtain the output value. Notice that this value is $f(x_1,\ldots,x_m,0)$ and is used to define $f(x_1,\ldots,x_m,1)$, so we keep it in a register z. Now, to compute $f(x_1,\ldots,x_m,1)$, we need only run the program h for input $(x_1,\ldots,x_m,0,z)$, since z contains $f(x_1,\ldots,x_m,0)$. The output can also be kept in register z, since the previous contents of register z is not required for $f(x_1,\ldots,x_m,2)$, so we may use the same register to store $f(x_1,\ldots,x_m,1)$. Iterate this process another y-1 times, we shall obtain $f(x_1,\ldots,x_m,y)$.

The program computes f is then clear (z is the output register), it is shown in Fig. 2. below:

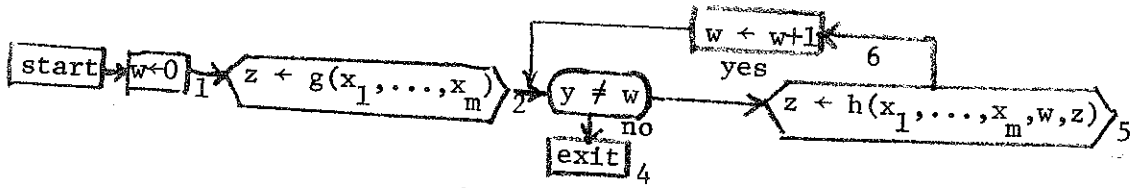


Fig. 2.

We have thus proved:

Theorem I. Primitive recursive functions are program computable.

2.2 Coding Function and Simultaneous Recursion

In this section, we introduce a particular coding from n-tuples of integers to integers. This coding scheme is reversible for that we can decode a coded integer into the original n-tuple. As shown in the following figure, to compute an n-tuple function Φ from N^n to N^n, instead of performing a direct computation based on the definition of Φ, we may code an n-tuple input into a single integer, then perform a computation for this integer on a corresponding single-variable function ϕ from N to N, finally decode the result back into an n-tuple value.

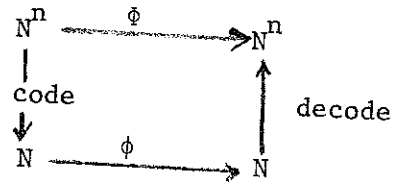


Fig. 3.

We now present our coding scheme. Let $P_0=2$, $P_1=3$, $P_2=5,\ldots,P_i=$ the (i+1)st prime number. In our coding scheme, we code an n-tuple $(x_1,\ldots,x_n)$ into a single number $g(x_1,\ldots,x_n)$, where

$$g(x_1,\ldots,x_n) = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \ldots P_{n-1}^{x_n} .$$

This coding function is certainly one to one. On the other hand, we know every positive integer y has a unique factorization into prime powers:

$$y = P_0^{a_0} P_1^{a_1} \ldots P_k^{a_k} .$$

Let $(y)_i$ denote the exponent a_i in this factorization. If y=1, $(y)_i$ is 0 for all i. If y=0, we arbitrarily let $(y)_i=0$ for all i. It is clear that $(y)_i$ serves the purpose as our decoding function. That is,

$$g((y)_0,(y)_1,\ldots,(y)_{n-1}) = y.$$

Notice that the coding function is by no means unique. Any computable function g could serve the purpose as long as the following conditions are satisfied:

(a) g is one to one.

(b) There is a unique procedure which determines whether any given positive integer y is in the range of g, and, if y is in the range of g,

(c) There is a procedure to decode y into the n-tuple $(x_1,\ldots,x_n)$ such that $g(x_1,\ldots,x_n)=y$.

We now show that our coding function and decoding functions are primitive recursive and thus computable. To see this, we need to introduce some other primitive recursive functions:

(1) Define a function "Prime":

$$Pr(x) = \begin{cases} 1 & \text{if x is a prime number,} \\ 0 & \text{otherwise.} \end{cases}$$

To determine whether a positive integer x is a prime or not, we simply count the number of divisors that x has. If the number of divisors of x is exactly two, then x is a prime number. Otherwise, x is not a prime number. Recall that the divisor function is:

$$D(i,x) = \begin{cases} 1 & \text{if } i \text{ is a divisor of } x, \\ 0 & \text{otherwise.} \end{cases}$$

So, the number of divisors of x is then $\sum_{i=0}^{x-1} D(i+1,x)$.

Thus, we have

$$Pr(x)=(1 \dot{-} sg(\sum_{i=0}^{x-1} D(i+1,x) \dot{-} 2))\cdot sg(x)\cdot sg(|x-1|)$$

The last item $sg(x)\cdot sg(|x-1|)$ inserted there is to make sure that $Pr(0) = 0$, and $Pr(1) = 0$. Pr(x) is defined in terms of composition and bounded summation from primitive recursive functions and thus is primitive recursive.

(2) Given a function f, we define a new function g in terms of f as follows:

$$g(x,z) = \begin{cases} \text{The smallest integer } j[j < z \text{ and } f(j,x)=1] \text{ if such } j \text{ exists.} \\ z, \qquad \text{otherwise.} \end{cases}$$

We shall use the notation

$$\mu\, j_{(j < z)}\, [f(j,x) = 1]$$

to represent such function, and we say g is defined from f by <u>bounded minimization</u> (or <u>bounded u-operator</u>).

If f is a primitive recursive, we now show that g will also be primitive recursive. For any x and y, the function

$$\prod_{i=0}^{y} (sg(|f(i,x)-1|))$$

is 1 if $f(i,x) \neq 1$ for all $0 \leq i \leq y$, and 0 if $f(i,x) = 1$ for some $0 \leq i \leq y$.

Case 1: If there exists a j, $j < z$, such that $f(j,x) = 1$ and $f(i,x) \neq 1$ for $i < j$, the above function is 1 if $0 \leq y \leq j-1$, and the above function is 0 if $j \leq y < z$. So the following function

$$\sum_{y=0}^{z-1} \left(\prod_{i=0}^{y} (sg\,(|f(j,x) - 1|)) \right)$$

is j.

Case 2: If for all j, $j < z$, $f(j,x) \neq 1$. In this case, the function

$$\sum_{y=0}^{z-1} \left(\prod_{i=0}^{y} (sg(|f(i,x)-1|)) \right)$$

is $\underbrace{1 + 1 + \ldots + 1}_{z} = z$.

We have thus shown that

$$\mu j_{(j < z)}\,[f(j,x) = 1] = \sum_{y=0}^{z-1} \left(\prod_{i=0}^{y} sg(|f(i,x) - 1|) \right).$$

Now we are able to prove P_i, the (i+1)st prime number is a primitive recursive function. Consider the number

$$P_i! + 1 = 1 \cdot 2 \cdot 3 \cdot 4 \cdots P_i + 1.$$

This number is either a prime or is not a prime. If this number is not a prime, then it has a prime factor which is greater than 1 but less than itself. Since all the prime numbers $P_0, P_1, \ldots, P_i$ are not a factor of $P_i! + 1$. We deduce that there exists at least a prime number which is greater than P_i but less than $P_1! + 1$. Now we can define the function P_i in terms of recursion:

$$P_0 = 2$$

$$P_{i+1} = \mu j_{(j \leq P_i! + 1)}\,[sg(j \dot{-} P_i) \cdot P(j) = 1].$$

Here, we define $P_0=2$, and P_{i+1} is the first number j such that j is greater than P_i (i.e. $sg(j \dot{-} P_i) = 1$) and j is a prime (i.e. $Pr(j) = 1$).

Next, we like to define $(x)_i$ in terms of primitive recursive rules. Remember $(x)_i$ is defined to be the exponent of P_i when x is factored into its prime powers. In other words, we simply search for the largest j such that P_i^j is a factor of x, or a j such that P_i^j is a factor of x and P_i^{j+1} is not a factor of x. So we get:

$$(x)_i = \mu j_{(j < x)}[(1 \dot{-} sg(rm(p_i^j,x))) \cdot sg(rm(P_i^{j+1},x))]= 1] \cdot$$

This completes the proof of the assertion that our coding and decoding functions are both primitive recursive.

We now consider an example, in which a system of recursively defined functions can be shown to be primitive recursive via the coding and decoding scheme.

Example: Consider the following two functions defined by recurrence equations.

$$\begin{cases} f_1(0) = a_1 \\ f_2(0) = a_2 \end{cases}$$

$$\begin{cases} f_1(x+1) = f_1(x)+f_2(x) \\ f_2(x+1) = f_1(x) \cdot f_2(x) \end{cases}$$

It is clear that these two equations determine the function values of both f_1 and f_2. But it is not quite clear that they are primitive recursive.

Let us define a new function $\underline{f}$ as follows:

$$\underline{f}(0) = 2^{a_1}.3^{a_2}$$

$$\underline{f}(x+1) = 2^{(\underline{f}(x))_0 + (\underline{f}(x))_1}.3^{(\underline{f}(x))_0 \cdot (\underline{f}(x))_1}$$

We can see that $\underline{f}$ is a primitive recursive for it is defined by recursion from known primitive recursive functions.

Claim that for x, $f_1(x) = (\underline{f}(x))_0$, and $f_2(x) = (\underline{f}(x))_1$. We prove this claim by induction on x:

Base: $f_1(0) = (\underline{f}(0))_0 = (2^{a_1} \cdot 3^{a_2})_0 = a_1$

$f_2(0) = (\underline{f}(0))_1 = (2^{a_1} \cdot 3^{a_2})_1 = a_2$

Induction step: Assume $f_1(x) = (\underline{f}(x))_0$ and $f_2(x) = (\underline{f}(x))_1$

$f_1(x+1) = (\underline{f}(x+1))_0 = (\underline{f}(x))_0 + (\underline{f}(x))_1 = f_1(x) + f_2(x)$,

and $f_2(x+1) = (\underline{f}(x+1))_1 = (\underline{f}(x))_0 \cdot (\underline{f}(x))_1 = f_1(x) \cdot f_2(x)$.

It thus follows that both f_1 and f_2 are primitive recursive.

This example provides a new rule for obtaining functions from given functions.

Definition. Let $g_1,\ldots,g_m$ be functions of m variables; and $h_1,\ldots,h_n$ be functions of (m+1)+n variables. The functions $f_1,\ldots,f_n$ of m+1 variables are said to be defined by simultaneous recursion from $g_1,g_2,\ldots,g_m,h_1,h_2,\ldots,h_n$, if for all $x_1,\ldots,x_{m+1}$

$$f_j(x_1,\ldots,x_m,0) = g_j(x_1,\ldots,x_m),$$

$$f_j(x_1,\ldots,x_m,x_{m+1}+1) = h_j(x_1,\ldots,x_{m+1},f_1(x_1,\ldots,x_{m+1}),\ldots,f_n(x_1,\ldots,x_{m+1}))$$

for j = 1,2,...,n.

To see that the set of functions $f_1,\ldots,f_n$ defined above are primitive recursive if g_i's and h_i's are primitive recursive, we define a new function $\underline{f}$:

$$\underline{f}(x_1,\ldots,x_m,0) = P_0^{g_1(x_1,\ldots,x_m)} \cdot P_1^{g_2(x_1,\ldots,x_m)} \cdot \ldots P_{n-1}^{g_n(x_1,\ldots,x_m)}$$

$$\underline{f}(x_1,\ldots,x_m,x_{m+1}+1) = P_0^{h_1(x_1,\ldots,x_{m+1},(\underline{f}(x_1,\ldots,x_{m+1}))_0,\ldots,(\underline{f}(x_1,\ldots,x_{m+1}))_{n-1})}$$

$$\vdots$$

$$\cdot P_{n-1}^{h_n(x_1,\ldots,x_{m+1},(\underline{f}(x_1,\ldots,x_{m+1}))_0,\ldots,(\underline{f}(x_1,\ldots,x_{m+1}))_{n-1})}$$

$\underline{f}$ is defined in terms of recursion from known primitive recursive functions.

So $\underline{f}$ is primitive recursive. Furthermore, we observe that each function f_i, $1 \leq i \leq n$, can be obtained by

$$f_i(x_1,\ldots, x_{m+1}) = (\underline{f}(x_1,\ldots,x_{m+1}))_{i-1},$$

In other words, we have proved:

Theorem II. The set of functions defined by simultaneous recursion from primitive recursive functions are primitive recursive.

2.3 The Equivalence of Primitive Recursive and Loop Program Computable Functions

In this section, we define a class of somewhat limited programs, called the loop program. We shall see that the class of functions computed by the loop programs is exactly the same as the class of primitive recursive functions.

Let us redraw the program in Fig. 2. as follows:

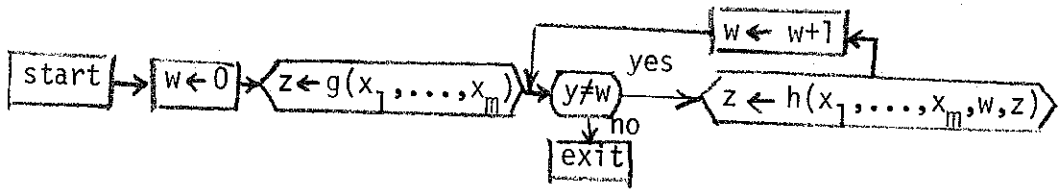


Fig. 4.

With input $(x_1,\ldots,x_m,y)$ the loop is run through precisely y times. This is because the function defined in terms of recursion has the property that one knows in advance (by the value of y) how many times the iteration will have to be done. We shall make this special feature as a characterization of a class of programs and hope that each primitive recursion function is computable by a program in this class, and vice versa. Let us define a new instruction from the program machine:

"loop"* loop y,m Repeat y times the body of the loop (i.e. the sequence of instructions from the one immediate followed to the one with instruction number m). If y contains 0, go to instruction m+1. The contents of y remains unchanged in any case.

*This instruction corresponds to the "DO" statement in FORTRAN.

As an example, consider a program which computes the predecessor of x, i.e. $z \leftarrow x \dot{-} 1$.

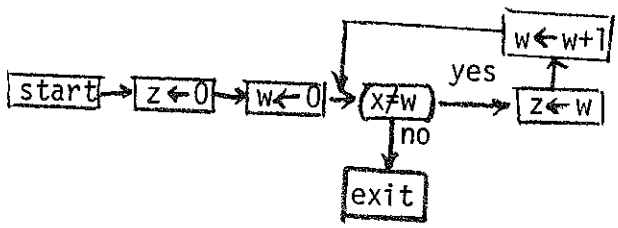


Fig. 5.

Using "loop" instruction, we have the following program listing:

```
1 :     z ← 0
2 :     w ← 0
3 :     loop x,5
4 :     z ← w
5 :     w ← w+1
6 :     exit
```

Fig. 6.

We shall investigate the functions that can be computed by programs limited to the following instructions: $x_i \leftarrow 0$, $x_i \leftarrow x_i+1$, "loop", and "exit".

Definition L, the class of loop programs, is defined as follows:

(1) L contains programs which consist of instructions of the form: $x_i \leftarrow 0$, $x_i \leftarrow x_i+1$, and trailed by an exit instruction.

(2) If P is a loop program with n instructions and y does not occur in P, then the following program is also a loop program:

```
loop y, n
P'
```

Where P' is the program P with its instruction numbers increased by 1. And instructions appeared as "loop x:m" in P are changed to "loop x:m+1.

(3) If programs P_1 and P_2 are in L, so is the concatenation of P_1 and P_2. Where the concatenation of P_1 and P_2 is formed by deleting the trailing exit instruction from P_1 and attaching

P_2' to it; P_2' is obtained from P_2 with the following modification: If P_1 has n instructions, then all the instruction numbers of P_2 are increased by n-1, and the instructions appeared in P_2 as "loop y:m" are changed to "loop y:m+n-1".

(4) No other program is in L.

Notice that a loop program has only one exit instruction, namely, the last instruction.

Definition

A function is loop-program computable if there exists a loop program that computes it.

We now give a characterization of loop-program computable functions.

Theorem III. A function is loop-program computable if and only if it is a primitive recursive function.

Proof. ⇐ The programs constructed in Section 2.1 to establish the computability of primitive recursive functions are all loop programs. The theorem thus follows directly.

⇒ We prove it by induction on the structure of loop programs.

(1) Clearly, all loop programs containing only instructions of the type $x_i \leftarrow 0$, $x_i \leftarrow x_i+1$ and trailed by exit instruction compute primitive recursive functions.

(2) Let us assume a program P of n instructions with m registers $x_1, x_2, \ldots, x_m$ computes m primitive recursive functions, one for each register,

$$f_1(x_1,\ldots,x_m),\ f_2(x_1,\ldots,x_m),\ldots,f_m(x_1,\ldots,x_m).$$

Then the program:

loop y, n

P'

defined in the above definition computes the functions $g_1(x_1,\ldots,x_m,y)$, $g_2(x_1,\ldots,x_m,y)$, $g_m(x_1,\ldots,x_m,y)$ at these registers, where

$$g_i(x_1,\ldots,x_m,0) = x_i,$$

$$g_i(x_1,\ldots,x_m,y+1)=f_i(g_1(x_1,\ldots,x_m,y),\ldots,g_m(x_1,\ldots,x_m,y)),$$

for i = 1,2,...,m. These functions, being defined from primitive recursive functions by simultaneous recursion, are therefore primitive recursive. The function for the register y is the identify function and thus is primitive recursive.

(3) We assume both programs P_1 and P_2 use registers $x_1,\ldots,x_m$ and each of them computes m primitive recursive functions, one for each register. Let the functions computed by P_1 be denoted as $f_1(x_1,\ldots,x_m),\ldots,f_m(x_1,\ldots,x_m)$ and the functions computed by P_2 be denoted by $g_1(x_1,\ldots,x_m),\ldots,g_m(x_1,\ldots,x_m)$. Then the concatenation of P_1 and P_2 computes m primitive recursive function at these registers.

$$h_i(x_1,\ldots,x_m)= g_i(f_1(x_1,\ldots,x_m),f_2(x_1,\ldots,x_m),\ldots,f_m(x_1,\ldots,x_m))$$

for i = 1,2,...,m. These functions, being defined by composition of two primitive recursive functions, are therefore, primitive recursive. Q.E.D.

Example. Consider the following loop program. The proof for the above theorem indicates a step-by-step method to find the functions computed by it. In this example, some instructions are not part of the instructions used in the definition of a loop program. However it does illustrate the concepts behind the theorem.

```
1 :   x2 ← 1
2 :   x3 ← 0
3 :   loop x1,5
4 :   x2 ← x2(x3+1)
5 :   x3 ← x3+1
6 :   exit
```

Solution. Let us first analyze the body of the "loop" at instruction number 3. It is easy to see that the functions defined at registers x_2 and x_3 are

$$f_2(x_2,x_3) = x_2 \cdot (x_3+1),$$
$$f_3(x_2,x_3) = x_3+1,$$

respectively. Following the proof of the theorem, the loop computes $g_2(x_1,x_2,x_3)$, $g_2(x_1,x_2,x_3)$, $g_3(x_1,x_2,x_3)$ at registers x_1,x_2,x_3, where

$$g_1(x_1,x_2,x_3) = x_1,$$

$$\begin{cases} g_2(0,x_2,x_3) = x_2, \\ g_3(0,x_2,x_3) = x_3 \end{cases}$$

$$\begin{cases} g_2(y+1,x_2,x_3) = f_2(g_2(y,x_2,x_3),g_3(y,x_2,x_3)) \\ \qquad\qquad = g_2(y,x_2,x_3) \cdot (g_3(y,x_2,x_3)+1) \\ g_3(y+1,x,x) = f_3(g_2(y,x_2,x_3),\ g_3(y,x_2,x_3)) \\ \qquad\qquad = g_3(y,x_2,x_3)+1 \end{cases}$$

Solving these two recurrence equations by an inductive analysis, we have:

$$g_2(y,x_2,x_3) = x_2(x_3+1)\ (x_3+2)\ldots(x_3+y)$$
$$g_3(y,x_2,x_3) = x_3+y$$

Now the original program is equivalent to the following one:

1 : $x_2 \leftarrow 1$

2 : $x_3 \leftarrow 0$

3 : $x_1 \leftarrow x_1$

4 : $x_2 \leftarrow x_2(x_3+1)\ (x_3+2)\ldots(x_3+x_1)$

5 : $x_3 \leftarrow x_3+x_1$

6 : exit

Which computes the functions at registers x_1,x_2, and x_3 as follows:

$$h_1(x_1,x_2,x_3) = x_1$$

$$h\ (x_1,x_2,x_3) = 1(0+1)\ (0+2)\ldots(0+x_1) = x_1!$$

$$h_3(x_1,x_2,x_3) = 0+x_1 = x_1$$

They are known primitive recursive functions.

Exercise

1. Give primitive-recursive definitions for

   a. $f(x) = x!$

   b. $f(0) = 0$, $f(x) = x^{\cdot^{\cdot^{\cdot^{x}}}}\}x$, e.g. $f(3) = 3^{3^3} = 3^{27}$.

   c. $f(x,0) = x$, $f(x,y) = 2^{\cdot^{\cdot^{\cdot^{2^{x}}}}}\}y$, e.g. $f(x,2) = 2^{(2x)}$.

   d. $f(0) = 2$, $f(1) = 3$, $f(x) = (x + 1)$st prime numbers.

2. If g is primitive recursive, show f defined below is also primitive recursive.

   $f(x,0) = g(x)$, $f(x,y + 1) = f(f(x,y),y)$

3. Let $a(x,y)$ be the Ackermann's function. We know that $a(0,y) = y + 1$ is the successor function.

   a. What is the value of $a(1,y)$, $a(2,y)$, $a(3,y)$ for a given y?

   b. Prove that for any fixed k, $a(k,y)$ is a primitive recursive function of y.

   c. Show that for any primitive recursive function g, there exists a constant c such that $a(x,x) > g(x)$ for $x \geq c$.

   This implies that the Ackermann's function can not be primitive recursive.

4. Define an infinite set of functions $p_0(x)$, $p_1(x)$,... as follows:

   $p_0(x) = x + 1$.

   $p_1(x)$ is the function computed by the following loop program (x is both the input and the output variable):

```
1:  y ← x
2:  loop y,3
3:  x ← p0(x)
4:  exit
```

   and $p_k(x)$ is the function computed by the following loop program (x is both the input and the output variable):

```
1:  y ← x
2:  loop y,3
3:  x ← pk-1 (x)
4:  exit
```

a. Show that $p_0(x) = x + 1$, $p_1(x) = 2x$, $p_2(x) = x \cdot 2^x$.

b. Can you express $p_3(x)$ explicitly?

c. Define $A(x) = a(x,x)$ where $a(x,y)$ is the Ackermann's function, $B(x) = p_x(2)$. Relate these two functions $A(x)$ and $B(x)$.

5. The coding-decoding scheme in Section 2.2 uses large numbers. We may define a paring function $\tau: N^2 \to N$ motivated by Figure 7 below, which shows the value $\langle x,y \rangle$ for some pairs (x,y).

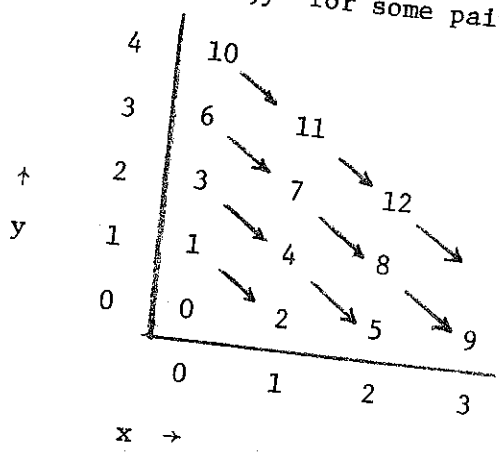


Figure 7

Note $\tau(x,y) = \left(\sum_{i=0}^{x+y} i \right) + x = x + 1/2\ (x + y)\ (x + y + 1)$, which is much smaller than the exponential coding given in Section 2.2.

a. $\tau(x,y)$ has different values for every pair of (x,y).

b. There exist primitive recursive decoding functions $\pi_1(z)$ and $\pi_2(z)$ for which

$$\pi_1(\tau(x,y)) = x, \text{ and } \pi_2(\tau(x,y)) = y$$

c. τ is one-one and onto..

Usually, we use the notation $\langle x,y \rangle$ to represent the coded number $\tau(x,y)$ for a pair (x,y). The coding function for 3-tuples can then be defined as:

$$\langle x,y,z \rangle = \tau(\langle x,y \rangle, z).$$

In general, the coding function for n-tuples can be defined in terms of the coding function of n-1 tuples.

$$\langle x_1,\ldots,x_n \rangle = \tau(\langle x_1,\ldots,x_{n-1} \rangle, x_n).$$

Reference

The discussion of primitive recursive functions given here is based on Mendelson [1964]. The characterization of primitive recursive functions in terms of loop programs may be found in Meyer and Ritchie [1967]. Ackermann's function represents a sort of simultaneous induction on two variables. Functions defined in this way are known as "double recursion" and they are discussed in Robinson [1948].

Chapter 3

PARTIAL RECURSIVE AND PROGRAM COMPUTABLE FUNCTIONS

3.1 Partial Recursive Functions

Consider the following program:

```
1:  z ← 0
2:  if z . z ≠ x then 4
3:  exit
4:  z ← z + 1
5:  goto 2
```

If x intially contains a square, then the above program will terminate and z will have the square root of x. On othe other hand, if x initially contains a number which is not a square, this program will never terminate. This program defines a partial function which is obviously computable. Observed that primitive recursive functions are total, hence the set of computable functions are not exhausted by the primitive recursive functions. In fact, some total computable functions are not primitive recursive either; among them, Ackerman's function is one example. In any case, we need to have some more terminiology in order to characterize the class of computable functions.

In section 2.2 we defined the rule of bounded minimization by which we obtained many useful functions. However, we proved that this rule does not generate functions beyond the realm of primitive recursive functions. We now introduce a similar rule, the minimization rule, by which we can obtain functions that are not primitive recursive.

Definition Let f be a total function of m + 1 variables. We say a partial function ϕ of m variable is defined from f by minimization (or μ-operator) if for all $x_1, \cdots, x_m$,

$$\phi(x_1,\ldots,x_m) = \begin{cases} \text{the smallest number } j \text{ such that } f(x_1, x_2, \ldots, x_m, j) = 1 \\ \qquad \text{if such an } j \text{ exists,} \\ \text{undefined} \qquad \text{otherwise.} \end{cases}$$

We use the notation $\phi(x_1,\ldots,x_m) = \mu j\ [f(x_1,\ldots,x_m,j) = 1]$.

We now define the class of partial recursive functions.

Definition: The class of partial recursive functions is the smallest class that can be obtained from the initial functions by any finite number of applications of composition, recursion, and minimization (μ-operator). In case a partial recursive function is total, we say it is general recursive, or simply recursive.

This differs from the definition of primitive recursive functions only by the addition of possible applications of the minimization rule. Hence, every primitive recursive function is recursive.

In this section, we shall show that the partial recursive functions are program-computable. The converse will be shown to be true in subsequent sections of this chapter. Notice that partial recursive functions obtained by composition or by recursion can be shown to be computable using similar arguments appeared in Chapter 2, in which primitive recursive functions are shown to be computable. So, to show that partial recursive functions are computable, it is now sufficient to establish the computability of those functions obtained by minimization.

Assume f is total and is computable, i.e. there is a program (the hexagonal box in Figure 1.) which computes f. The function ϕ defined from f by minimization is then computed by the program below (z is the output register):

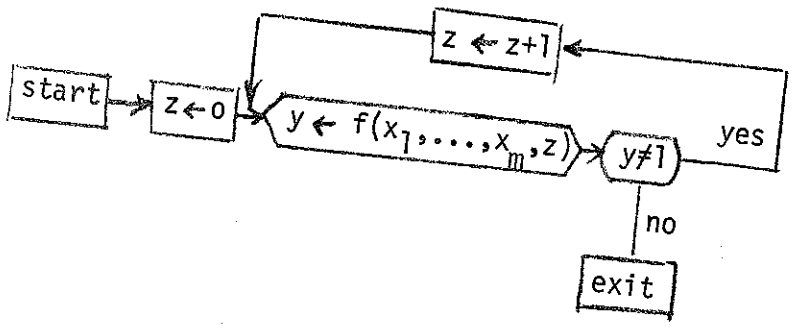


Figure 1.

Since f is total the program computing f will terminate and yield an output. If this output value is 1, then the above program will terminate and yield the correct output in register z. Otherwise, it will simply go on to test the next number in the sequence. If $f(x_1,\ldots,x_m,z)$ is not equal to 1 for any z, the above program will never terminate, $\phi(x_1,\ldots,x_m)$ is thus undefined. We now complete the proof of the theorem:

Theorem I. Partial recursive functions are program computable.

3.2 Gödel numbers for programs

The modern digital computers are usually capable of manipulating with strings of symbols. On the other hand, our program machine, a particular model of a digital computer, is dealing with numbers. That is, it has no capability to store or to operate on strings of symbols. However, this is no drawback because one can always code strings of symbols as numbers and then work with the coded numbers on the program machine. Such a process of coding strings of symbols into numbers is generally called Gödel numbering.

In the present section, we shall introduce a Gödel numbering for programs. With such a coding scheme programs represented by its coded number are themselves capable of being stored and operated on by a program machine. The Gödel numbering introduced here is based on the unique

factorization of numbers into prime powers. Of course, other Gödel numberings can also be used as long as coding programs into numbers and decoding numbers back to programs can effectively be achieved.

For simplicity, let us consider programs which are sequences of instructions from the original set. That is, programs consist of instructions of the following type:

(a) $x_i \leftarrow 0$
(b) $x_i \leftarrow x_i + 1$
(c) <u>if</u> $x_i \neq x_j$ <u>then</u> n
(d) <u>exit</u>

Furthermore, we may assume that a program uses m registers. They are denoted respectively by $x_1, x_2, \ldots, x_m$. We now describe our Gödel numbering. Recall that $p_0 = 2$, and p_i = the (i+1)st prime number.

(1) At each instant of the computation, let the contents of the m registers be $a_1, a_2, \ldots, a_m$. We code these numbers by

$$2^{a_1} \cdot 3^{a_2} \ldots p_{m-1}^{a_m}.$$

In this fashion the complete set of the contents of m registers is given by a single number.

(2) Each individual instruction is assigned a unique Gödel number as shown in the following figure.

Instruction	Gödel number
$x_i \leftarrow 0$	$2^{i+1}.3$
$x_i \leftarrow x_i + 1$	$2^{i+1}.5$
<u>if</u> $x_i \neq x_j$ <u>then</u> n	$2^{i+1}.7^{j+1}.11^{n+1}$
<u>exit</u>	13

Figure 2.

(3) Let us assume that the given program has n instructions and these instruction numbers are from 1 through n. Let b_i be the Gödel number of the instruction number i. Then the Gödel number of the program is defined as:

$$2^{b_1}.3^{b_2}.5^{b_3}\ldots P_{n-1}^{b_n}.$$

With these definitions, any program can be represented by just one single number. Although the Gödel number of even a very short program tends to be very large. For example, the Gödel number of the following program

1: $x_1 \leftarrow 0$
2: $x_1 \leftarrow x_1 + 1$
3: if $x_1 \neq x_2$ then 2
4: exit

is $2^{2^2.3}.3^{2^2.5}.5^{2^2}.7^{3^3}.11^{3^{13}}$, which is a tremendously large number. Nevertheless, given such a number all we are interested in is the program represented by it. With this coding scheme, one can easily decode a number and know right away whether the number represents a program, and obtains the program effectively in case that number does represent one.

3.3 Universal program machines

The program machines that we have been studying are not store-program computers for that each machine performs only one job. On the other hand, a store-program computer will be able to perform any job that can be specified by a program. The store-program computer takes a program and data for that program as input (of course, both the program and the data are described as strings of symbols in some specific form). Then it will perform operations on the data according to the instructions of the program, and it will generate the output if there is any. In other words, the store-program computer behaves exactly like the input program behaves and thus performs the job as the input program specifies.

In this section, we shall present a particular program machine, the universal program machine, which takes two Gödel numbers, one represents a program and the other represents the initial contents of the registers used by the program, and it will give exactly the output that the input program expected to produce. Thus, the universal program machine is the

counterpart of a store-program computer. The universal program machine can be thought of as a computer which takes a program written in machine language (represented by its Gödel number) and executes the instructions specified by the program.

The following picture gives the idea:

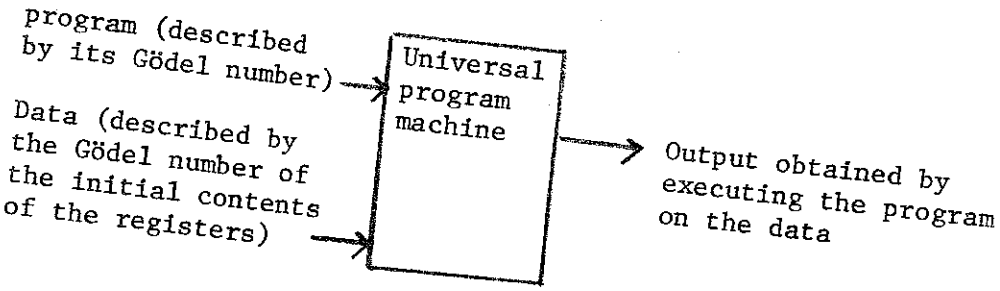


Figure 3.

The universal program machine uses four registers z, x, ℓ, and b:

z stores the Gödel number of a program to be executed.

x stores the Gödel number of the initial contents of the m registers used by the input program: $2^{a_1} 3^{a_2} \ldots P_{m-1}^{a_m}$

ℓ stores the instruction number of the instruction to be executed.

b stores the Gödel number of the instruction to be executed.

As an analogy, z is the memory which stores the machine language program. x is the memory which stores the data. ℓ is the instruction address register, and b is the buffer register.

We now define three primitive recursive functions:

(1) fetch (z,ℓ): this function will fetch an instruction (its Gödel number) specified by the ℓ register from the program z(z register contains the Gödel number of a program).

By our Gödel numbering scheme, we know

$z = 2^{b_1} \cdot 3^{b_2} \cdot \ldots \cdot P_{\ell-1}^{b_\ell} \ldots P_{n-1}^{b_n}$, so

fetch $(z,\ell) = (z)_{\ell-1}$,

which is a primitive recursive function.

(2) label (b,x,ℓ): this function will give the instruction number of the next instruction to be executed after the instruction b is executed. The current contents of the input data are specified in x.

To compute label (b,x,ℓ) we first see if b, the current instruction, is not a multiple of 7 (Please refer to Fig. 2). If so, output $\ell + 1$. Otherwise, find out the register x_i and x_j from b and then compare the contents of x_i and the contents of x_j. If they agree, output $\ell + 1$, otherwise, find out the exponent of 11 of b and output that number minus 1. The following program computes label (b,x,ℓ) (y is the output register. b is the Gödel number of the instruction to be executed, ℓ is the current instruction number):

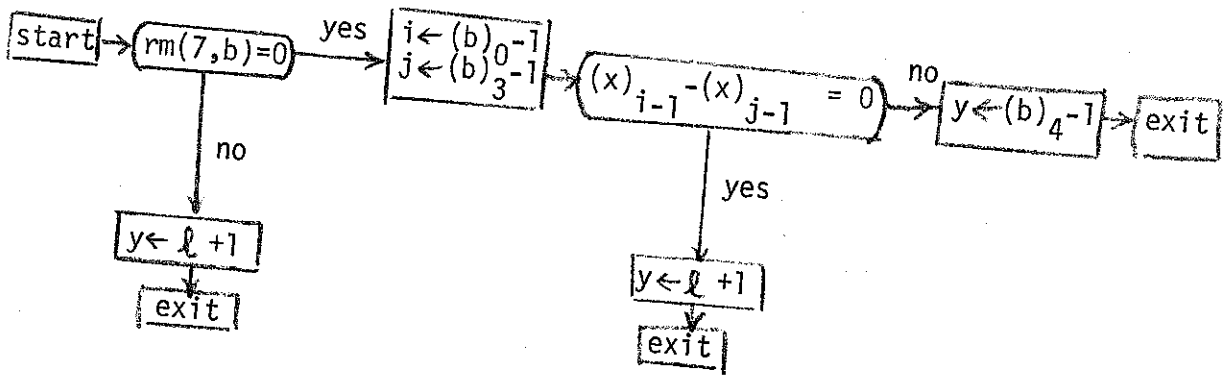


Figure 4.

We leave it as an exercise to show that label (b,x,ℓ) is primitive recursive.

(3) exec (b,x): This function will perform whatever the operations specified by the instruction b on x and output the resultant value of x.

To compute exec (b,x), let us first discuss how we perform operations on x.

(a) $x_i \leftarrow 0$: Simply repeatedly divide x by P_{i-1} until P_{i-1} is not a divisor of x.

(b) $x_i \leftarrow x_i + 1$: Simply multiply x by P_{i-1}.

(c) if $x_i \neq x_j$ then n: x remains unchanged

(d) exit: x remains unchanged.

Second, we check if 3 is a factor of b, if so, set $i = (b)_0 - 1$ and perform $x_i \leftarrow 0$. Otherwise see if 5 is a factor of b, if so, set $i = (b)_0 - 1$ and perform $x_i \leftarrow x_i + 1$. Otherwise do nothing.

The reader is encouraged to verify that exec (b,x) is also primitive recursive.

Now we are in a position to design the universal program machine. In terms of the functions fetch, label, and exec discussed above, the program U is composed as shown in Figure 5.

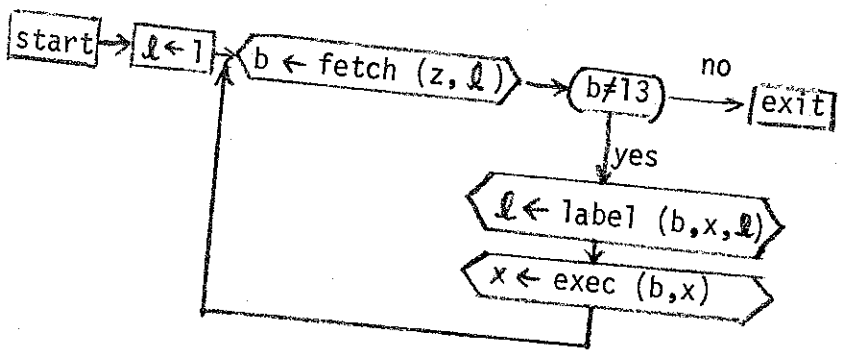


Figure 5.

Let us discuss the behavior of U in terms of an example. Let register z initially contain the Gödel number of the following program:

1: $x_1 \leftarrow 0$
2: $x_1 \leftarrow x_1 + 1$
3: if $x_1 \neq x_2$ then 2
4: exit

Let (a_1,a_2) be the input to the above program, then register x initially contains the Gödel number of the input value $2^{a_1} \cdot 3^{a_2}$ Assume $a_2 > 0$.

In the first step ℓ is set to 1, which indicates the instruction to be executed is instruction number 1. Then the loop of U started.

The first step in the loop is to compute fetch(z,ℓ) and assign its value

to b. This step essentially fetch the first instruction "$x_1 \leftarrow 0$" and place its Gödel number $2^2.3$ in b. Now this number is not equal to 13 (i.e. this instruction is not the exit instruction), so U performs $\ell \leftarrow$ label $(2^2.3, 2^{a_1}.3^{a_2}, 1)$ which increase ℓ to 2. After the instruction $x \leftarrow$ exec $(2^2.3, 2^{a_1}.3^{a_2})$ is completed by U, x contains 3^{a_2}.

Now U starts the second run through the loop and computes b $\leftarrow$ fetch $(z,2)$ which leaves b the number $2^2.5$ (i.e. fetch the second instruction $x_1 \leftarrow x_1 + 1$). Again, b is not equal to 13 so U performs $\ell \leftarrow$ label $(2^2.5, 3^{a_2}, 2)$ which increase ℓ to 3. After the execution of $x \leftarrow$ exec $(2^2.5, 3^{a_2})$, x becomes $2^1.3^{a_2}$.

The third time U runs through the loop, it will fetch the next instruction "if $x_1 \neq x_2$ then 2" and place its Gödel number $2^2.7^3.11^3$ in b. Again, b is not equal to 13 so U performs $\ell \leftarrow$ label $(2^2.7^3.11^3, 2^1.3^{a_2}, 3)$ which set ℓ to 2. After the execution of $x \leftarrow$ exec $(2^2.7^3.11^3, 2^1.3^{a_2})$, x remains to be $2^1.3^{a_2}$.

The same process as the last two times through the loop of U will be executed over and over until x contains $2^{a_2}.3^{a_2}$. At this moment, the execution of the instruction $\ell \leftarrow$ label $(2^2.7^3.11^3, 2^{a_2}.3^{a_2}, 3)$ by U will set ℓ to 4. Then, the next run through the loop of U computes b $\leftarrow$ fetch $(z,4)$ which sets b to be 13. The final step will discover that b is equal to 13 and stop the computation. The output of U is in register x which is equal to $2^{a_2}3^{a_2}$.

From this example, it is now clear that every time the loop of U is carried through, one instruction of the input program is executed by U. Thus, U performs exactly as the input program specifies. State this result as a theorem, we have:

Theorem II There exists a program machine U such that for every m and for any program with m registers, the program terminates on input $<a_1,\ldots,a_m>$ with output $<d_1,\ldots,d_m>$ if and only if U terminates on input $<z, 2^{a_1}.3^{a_2}\ldots p_{m-1}^{a_m}>$ with

output $\langle z, 2^{d_1}.3^{d_2}\ldots p_{m-1}^{d_m}\rangle$, where z is the Gödel number of the input program. The program machine U is called a universal program machine.

3.4 Computable Functions are Partial Recursive

In section 3.1 we showed that partial recursive functions are computable, we now show the converse. Recall that the universal program machine U is able to imitate the computation for any program on any input. Initially, the register x contains the coded m-tuple, i.e.

$$x^{(1)} = 2^{a_1}.3^{a_2}\ldots p_{m-1}^{a_m}$$

and the content of b at this moment is

$$b^{(1)} = \text{fetch } (z,1),$$

where z is the Gödel number of the input program to U. Each time the loop is carried through, the contents of x and b are changed to new values. Notice that the new values of x and b are completely determined by z, the Gödel number of input program, and the current values of x and b. Let $x^{(j)}$, $b^{(j)}$, and $\ell^{(j)}$ be the values of register x, b, and ℓ at the beginning of the jth cycle through the loop of U, then at the beginning of the next cycle:

$$b^{(j+1)} = \text{fetch } (z, \ell^{(j)})$$

$$x^{(j+1)} = \text{exec } (b^{(j)}, x^{(j)}),$$

and $$\ell^{(j+1)} = \text{label } (b^{(j)}, x^{(j)}, \ell^{(j)}))$$

We can thus effectively generate a finite or infinite sequence of pairs

$$\langle b^{(1)}, x^{(1)}\rangle, \langle b^{(2)}, x^{(2)}\rangle, \ldots .$$

This sequence is finite if it contains a pair $\langle b^{(s)}, x^{(s)}\rangle$ such that $b^{(s)}$ is equal to 13. In this case $x^{(s)}$ contains the output associated with the input $x^{(1)}$. On the other hand, if for all pairs $\langle b^{(j)}, x^{(j)}\rangle$, $b^{(j)}$ is never equal to 13 then this sequence is infinite. In this case the output associated with the input $x^{(1)}$ is undefined. With the assist of the μ-operator the output

associated with the input $x^{(1)}$ is determined as:

$$x^{(\mu j[b^{(j)} = 13])}$$

Stating this result as a theorem, we have

Theorem III Let $x_1, \ldots, x_m$ be the registers used in a program, and let $\phi_k(x_1, \ldots, x_m)$ be the partial function computed by this program with output variable x_k. Then ϕ_k is a partial recursive function.

Proof. Define three functions g_1, g_2 and g_3 as follows:

$$\begin{cases} g_1(z,x_1,\ldots,x_m,0) = \text{fetch}\ (z,1) \\ g_2(z,x_1,\ldots,x_m,0) = 2^{x_1}.3^{x_2}.\ \ldots\ p_{m-1}^{x_m} \\ g_3(z,\ x_1,\ldots,x_m,0) = 1 \end{cases}$$

$$\begin{cases} g_1(z,x_1,\ldots,x_m,j+1) = \text{fetch}(z,g_3(z,x_1,\ \ldots,\ x_m,j)) \\ g_2(z,x_1,\ldots,x_m,j+1) = \text{exec}(g_1(z,x_1,\ldots,x_m,j),g_2(z,x_1,\ldots,x_m,j)) \\ g_3(z,x_1,\ldots,x_m,j+1) = \text{label}(g_1(z,x_1,\ldots,x_m,j),g_2(z,x_1\ldots,x_m,j),g_3(3,x_1,\ldots, \\ x_m,j)) \end{cases}$$

Being defined by simultaneous recursion from primitive recursive functions, g_1, g_2, and g_3 are therefore primitive recursive.

Next let

$$t(z,x_1,\ldots,x_m,j) = 1\dot{-}sg(|g_1(z,x_1,\ldots,x_m,j) - 13|), \text{ and}$$

$$p(z,x_1,\ldots,x_m,j) = (g_2(z,x_1,\ldots,x_m,j))_k,$$

t and p are obtained from composition of primitive recursive functions and thus are primitive recursive.

We now let the Gödel number of the given program be z, then we have

$$\phi_k(x_1,\ldots,x_m) = p(z,x_1,\ldots,x_m,\ \mu j[t(z,x_1,\ldots,x_m,j)=1]).$$

for all $x_1,\ldots,x_m$. It follows that ϕ_k is partial recursive. Q.E.D.

Corollary III (a) Program computable functions are partial recursive.

Theorem III says that any program computable function can be defined in terms of one application of the μ-operator on primitive recursive functions. This is generally known as Kleene Normal Form Theorem. The function t is a 0-1 valued function and is generally known as Kleene T-predicate.

Corollary III (b) (Kleene Normal Form Theorem). For every m, there exists two primitive recursive functions t and p such that for every partial recursive function ψ, there exists a number z with the property that

$$\psi(x_1,\ldots,x_m) = p(z,x_1,\ldots,x_m\ ,\ \mu j[t(z,x_1,\ldots,x_m,\ j) = 1])$$

for all $x_1,\ldots,x_m$.

Proof. Since ψ is partial recursive, so there exists a program which computes ψ. Let z be the Gödel number of the program. So, the result follows from Theorem III. Q.E.D.

3.5 Complexity of Primitive Recursive Functions.

While it is important to understand the capabilities and limitations of a computing device, there are other questions of vast concern to a computer scientist. For example, given that a function is program computable, an immediate question arises as to how difficult the function is to compute, as measured by some criterion such as the amount of time or storage required. It is clear that the same function can be computed by different programs. The time it takes to execute each of these programs is usually different. So we shall talk about the complexity of programs. In this section, we investigate the complexity measured by the amount of time required to compute primitive recursive functions.

For a given program P*, let $T_P(x)$ be the number of instructions executed by this program with input x if the program halts, and be undefined otherwise. $T_P(x)$ is considered as the amount of time (we assume

*For simplicity, we only consider programs with one input register.

every instruction takes the same amount of time to be executed) required by program P with input x. The following theorem characterizes the primitive recursive functions in terms of how difficult they are to compute.

Theorem IV. A function f is primitive recursive if and only if there is a program P which computes f and $T_P(x)$ is bounded by a primitive recursive function for all x.

Proof.

$\Rightarrow$ Assume f is primitive recursive. By Theorem 2-III there exists a loop program P that computes f. Assume y is a register not appearing in P. We create a new program Q as follows:

a) Add the instruction "y←0;" at the beginning of program P;

b) Insert the instruction "y←y+1;" before each instruction of program P.

The new program Q obviously computes $T_P(x)$ with y to be the output register. Program Q is also a loop program. So $T_P(x)$ is itself a primitive recursive function.

$\Leftarrow$ Assume f is computed by a program P and $T_P(x)$ is bounded by a primitive recursive function g(x). Now we simulate program P by the universal program machine constructed in section 3.3 Each time the loop of the universal program is carried through, one instruction of P is executed and vice versa. Since $T_P(x)$ is bounded by g(x), so the number of times that the loop of the universal program will be carried through is also bounded by g(x). Following the same arguments as in section 3.4, we can show that there exist two primitive recursive functions p and t such that

$$f(x) = p(z,x,\mu j_{(j<g(x))}[t(q,x,j)=]).$$

for all x, where z is the Gödel number of program P. In other words, f is primitive recursive since it is obtained by bounded μ-operator from primitive recursive functions. Q.E.D.

The theorem says f is primitive recursive if and only if it can be computed by a program using a primitive recursive number of steps. So it implies that the computation time for any nonprimitive recursive function must grow faster for some x than any primitive recursive function such as

$$g(x) = x^{x^{\cdot^{\cdot^{x}}}} \quad (x \text{ copies of } x).$$

Hence nonprimitive recursive functions cannot in general be practically computed.

d. Define a function

$$f(i) = \begin{cases} 1 & \text{if } p_i(i) \text{ is even,} \\ 2 & \text{if } p_i(i) \text{ is odd.} \end{cases}$$

Is f recursive? Is it primitive recursive? Why?

4. Define a function

$$\psi(x_1,\ldots,x_m) = \begin{cases} \phi_1(x_1,\ldots,x_m) & \text{if } \phi_1(x_1,\ldots,x_m) \text{ is defined,} \\ \vdots & \\ \phi_k(x_1,\ldots,x_m) & \text{if } \phi_k(x_1,\ldots,x_m) \text{ is defined,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Show that if $\phi_1,\ldots,\phi_k$ are partial recursive, and if, for any $x_1,\ldots,x_m$, at most one of $\phi_1(x_1,\ldots,x_m),\ldots,\phi_k(x_1,\ldots,x_m)$ is defined, then ψ is partial recursive.

Reference

The way that Gödel numbers are assigned and the universal program machine is constructed is largely from Engeler [1973]. The characterization of the partial recursive functions using the T-predicate and one application of the μ-operator is shown in Kleene [1943]. Superb discussions on equivalence of various models for effective computability can be found in Minksy [1967]. The complexity of loop programs is duscussed in Meyer and Ritchie [1967].