

Chapter 3

Programs and Machines

In this chapter, we focus on another way of formalizing the concept of effective calculability, namely register machine programs. Our first goal is to show that all general recursive partial functions are also computable by register machines. This fact will allow us to apply our work in Chapter 2 to see that a great many everyday functions are register-machine computable. Using this, we will be able to construct a universal program, that is, a program to compute the partial function $\Phi(w, x)$ = the result of applying the program coded by w to the input x .

Register machines

Recall from Chapter 1 that a register machine program is a finite sequence of instructions of the following types:

- I r (where r is a numeral for a natural number). “Increment r .” The effect of this instruction is to increase the contents of register r by 1. The machine then proceeds to the next instruction in the program (if any).
- D r (where r is a numeral for a natural number). “Decrement r .” The effect of this instruction depends on the contents of register r . If that number is non-zero, it is decreased by 1 and the machine proceeds *not* to the next instruction, but to the following one. But if the number in register r is zero, the machine simply proceeds to the next instruction. In summary, the machines tries to decrement register r and if it is successful then it skips one instruction.
- J q (where q is a numeral for an integer in \mathbb{Z}). “Jump q .” All registers are left unchanged. The machine takes as its next instruction the q th instruction following this one in the program (if $q \geq 0$), or the $|q|$ th instruction preceding this one (if $q < 0$). The machine halts if there is no such instruction in the program. An instruction of J 0 results in a loop, with the machine executing this one instruction over and over again.

Now suppose f is an n -place partial function on \mathbb{N} . Possibly there will be a program \mathcal{P} such that if we start a register machine (having all the registers to which \mathcal{P} refers) with x_1, \dots, x_n in registers $1, \dots, n$ and 0 in the other registers, and we apply program \mathcal{P} , then the following conditions hold:

- If $f(x_1, \dots, x_n)$ is defined, then the computation eventually terminates with $f(x_1, \dots, x_n)$ in register 0. Furthermore, the computation terminates by seeking a $(p + 1)$ st instruction, where p is the length of \mathcal{P} .
- If $f(x_1, \dots, x_n)$ is undefined, then the computation never terminates.

If there is such a program \mathcal{P} , we say that \mathcal{P} computes f . (Notice when we start a program running, there are three possibilities: (i) it might run forever; (ii) it might come to a “good” halt, by seeking the first instruction that isn’t there; (iii) it might come to a “bad” halt, either by trying to jump back to somewhere before the start of the program, or by trying to go forward to a non-existent instruction other than the first such one. In our definition of “ \mathcal{P} computes f ” we have elected to rule out bad halts. This will be a convenience in running programs end-to-end.)

For example, the addition function $x + y$ is computed by a register machine program given in Chapter 1. Moreover, in Chapter 1 we saw a few basic subroutines:

CLEAR	r		(clear register r)
MOVE	from r to s		(register r is cleared)
COPY	from r to s	using t	(register r is unchanged)

These subroutines have length 3, 7, and 15, respectively.

For a trivial example, the n -place constantly zero function is computed both by the empty program and by our three-line program to clear register 0:

→	D	0		Try to decrement 0.
└─	J	2	┌─	Go back and repeat.
	J	-2	└─	
			←	Halt.

(The arrows are to help us see what the program does.) Moreover, by adding some increment instructions, we can see that any total constant function is computed by some register program.

The 1-place successor function $S(x) = x + 1$ is computed by the program that moves the contents of register 1 (call this number [1]) to register 0, and then increments the number:

→	D	1		Take 1 from [1].
└─	J	3	┌─	Exit when zero.
	I	0	└─	
	J	-3		Repeat.
	I	0	←	Add 1 to [0].

The projection function I_n^k is computed by the seven-line program that moves the contents of register n to register 0.

Next, we want to show that the class of partial functions computed by register machine programs is closed under composition. That is, we want to know that whenever we have register machine programs computing f, g_1, \dots, g_n and

$$h(\vec{x}) = f(g_1(\vec{x}), \dots, g_n(\vec{x}))$$

then we can produce a program for h . This involves stringing several programs together. But care must be taken to be sure that one program does not trip

over garbage left by an earlier program, and does not erase data needed by a later program.

We know what it means for \mathcal{P} to compute f : When we provide \mathcal{P} with ideal conditions (the input in registers $1, \dots, k$, other registers empty), then \mathcal{P} will return (in register 0) the function value, if it is defined.

But we need a program that is less fussy. What if conditions are not ideal? Suppose the input is in registers r_1, \dots, r_k , where these are any k distinct numbers (not necessarily consecutive, and not necessarily in increasing order). And suppose we do not want to promise that the other registers are empty. Moreover, we want a program that does not erase or tamper with the contents of the first s registers, for some large number s —we want the information in those registers to be kept safe.

Here is what we want, formulated as a definition:

Definition: Suppose that f is a k -place partial function, \mathcal{Q} is a program, r_1, \dots, r_k are distinct natural numbers, and s and t are natural numbers. Then we say that \mathcal{Q} *computes f from r_1, \dots, r_k to t preserving s* if whenever we start a register machine (with enough registers) with program \mathcal{Q} and with a_1, \dots, a_k in registers r_1, \dots, r_k , then, no matter what is in the other registers initially, we have the following end results:

- If $f(a_1, \dots, a_k)$ is defined, then the computation eventually comes to a good halt (that is, it halts by seeking the first non-existent instruction, the $(q+1)$ st instruction, where q is the length of \mathcal{Q}), with the value $f(a_1, \dots, a_k)$ in register t . Moreover, the first s registers, registers $0, 1, \dots, s-1$, contain the same numbers they held initially, except possibly for register t .
- If $f(a_1, \dots, a_k)$ is undefined, then the computation never halts.

As a special case, we can say that if \mathcal{Q} computes f from $1, \dots, k$ to 0 preserving 0, then \mathcal{Q} computes f (in the sense defined originally). The converse is not quite true, because of the matter of whether registers are initially empty.

The following theorem says that we can have what the preceding definition asks for.

Lemma: Assume that the program \mathcal{P} computes the k -place partial function f . Let r_1, \dots, r_k be distinct natural numbers; let t and s be natural numbers. Then we can find a program \mathcal{Q} that computes f from r_1, \dots, r_k to t preserving s .

Proof: Let M be the largest *address* in \mathcal{P} (that is, the largest number such that some increment or decrement instruction in \mathcal{P} addresses register M). Probably $M > k$; if not then increase M to be $k+1$. Let j be the largest of the

numbers s, r_1, \dots, r_k . Here is program \mathcal{Q} :

```

COPY   $r_1$  to  $j + 1$  using  $j + 2$ 
COPY   $r_2$  to  $j + 2$  using  $j + 3$ 
...
COPY   $r_k$  to  $j + k$  using  $j + k + 1$ 
CLEAR  $j$ 
CLEAR  $j + k + 1$ 
CLEAR  $j + k + 2$ 
...
CLEAR  $j + M$ 
       $\mathcal{P}$  relocated by  $j$ 
MOVE  from  $j$  to  $t$                 (if  $j \neq t$ )

```

Here “relocated by j ” means that j is added to the address of all increment and decrement instructions. Thus the relocated program operates on registers $j, j + 1, \dots, j + M$ exactly as \mathcal{P} operated on registers $0, 1, \dots, M$. Since \mathcal{P} calculates f , the relocated program will leave $f(a_1, \dots, a_k)$, if defined, in register j . The program \mathcal{Q} then moves it to register t . Except for register t , the program leaves registers $0, 1, \dots, j - 1$ unchanged. The following map illustrates how \mathcal{Q} uses the registers:

register	0	untouched	
register	1	untouched	
	⋮		
register	$j - 1$	untouched	
register	j	output	
register	$j + 1$	input	
	⋮		
register	$j + k$	input	
register	$j + k + 1$	work space	
	⋮		
register	$j + M$	work space	⊥

The lemma will be a useful tool whenever we need to glue different programs together. As our first application of this lemma, we can show that the class of register-machine computable partial functions is closed under composition.

Theorem: The class of register-machine computable partial functions is closed under composition. That is, whenever we have register-machine programs that compute partial functions f, g_1, \dots, g_n from which h is obtained by composition, then we can make a program that computes the partial function h .

Proof. Suppose that the k -place partial function h is obtained by composition from f and g_1, \dots, g_n :

$$h(\vec{x}) = f(g_1(\vec{x}), \dots, g_n(\vec{x})).$$

Further suppose that we have programs that compute f, g_1, \dots, g_n . We want to make a program that computes h . Here it is:

- Calculate g_1 from $1, \dots, k$ to $k + 1$, preserving $k + 1$.
- Calculate g_2 from $1, \dots, k$ to $k + 2$, preserving $k + 2$.
- ...
- Calculate g_n from $1, \dots, k$ to $k + n$, preserving $k + n$.
- Calculate f from $k + 1, \dots, k + n$ to 0 , preserving 0 .

Here we rely on the lemma to provide the components of the program. Observe that for the program to halt, we need $g_1(\vec{x}), \dots, g_n(\vec{x})$ to be defined, and we need f to be defined at $\langle g_1(\vec{x}), \dots, g_n(\vec{x}) \rangle$. \dashv

For example, suppose that h is given by the equation

$$h(x, y, z) = f(g(z, y), 7, x)$$

and we have register programs for f and g . It follows from the preceding theorem that h is register-machine computable. We can write

$$h(x, y, z) = f(g(I_3^3(x, y, z), I_2^3(x, y, z)), K_7(x, y, z), I_1^3(x, y, z))$$

(where K_7 is a constant function) and apply the theorem twice, first to get $g(I_3^3(x, y, z), I_2^3(x, y, z))$ and then to get h . The moral of this example is that we can freely put the variables where we want, and apply composition with projection functions to justify what we have done. In particular, if

$$h(x_1, x_2, \dots, x_m) = f(_, _, \dots, _),$$

where each blank is filled by some x_i or some constant, then from a program for f we can obtain a program for h .

Still, this chapter has not yet produced a program for a single “interesting” function. For that, we need one more closure result: closure under primitive recursion. That is, we want to know that if h is obtained from f and g by primitive recursion

$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y + 1) &= g(h(\vec{x}, y), \vec{x}, y) \end{aligned}$$

and we have register programs for f and g , then we can get a program for h . Or in the case where \vec{x} is empty

$$\begin{aligned} h(0) &= m \\ h(y + 1) &= g(h(y), y) \end{aligned}$$

(for some number m) and we have a program for g , then we want to know that we can get a program for h .

It will then follow that all primitive recursive functions (in particular, the ones in Chapter 2) are register-machine computable. And so finally we will see that the class of register-machine computable functions includes much more than the simplistic examples we started with.

Theorem: The class of register-machine computable partial functions is closed under primitive recursion.

Proof. Assume that h is the partial $(n + 1)$ -place function obtained by primitive recursion from the partial functions f and g :

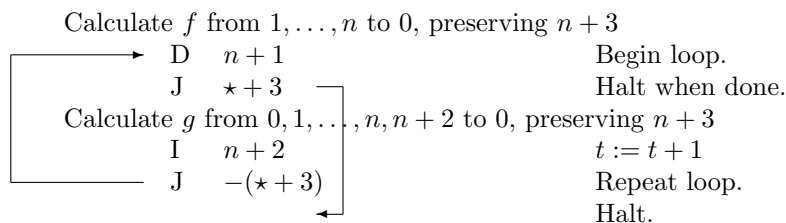
$$\begin{aligned} h(\vec{x}, 0) &= f(\vec{x}) \\ h(\vec{x}, y + 1) &= g(h(\vec{x}, y), \vec{x}, y) \end{aligned}$$

Assume that we have register programs for f and g . We need to make a program for h .

The program will start with x_1, \dots, x_n, y in registers $1, 2, \dots, n, n + 1$. The program will put $h(\vec{x}, t)$ in register 0 first for $t = 0$, then for $t = 1$, and so forth up to $t = y$. The number t is kept in register $n + 2$, which initially contains 0. The following map illustrates this usage:

register	0	$h(\vec{x}, t)$
register	1	x_1
	⋮	
register	n	x_n
register	$n + 1$	$y - t$
register	$n + 2$	t
register	$n + 3$	work space
	⋮	

Here is the program:



Here \star is the length of the program being used for g .

To see the correctness of this program, we establish the following:

Claim: Whenever we reach the D $n + 1$ instruction (at the beginning of the loop), after executing the loop k times,

- register 0 contains $h(\vec{x}, k)$,
- register $n + 1$ contains $y - k$,
- register $n + 2$ contains k .

The claim, stating “loop invariants,” is proved by induction (as is usual, for programs with loops) on k .

For $k = 0$, when we reach D $n + 1$ without having executed the loop at all, register 0 contains $f(\vec{x})$, which is $h(\vec{x}, 0)$, register $n + 1$ is untouched so it still contains y , and register $n + 2$ is still 0.

Now for inductive step. In the $(k + 1)$ st pass through the loop, we decremented register $n + 1$ (by the inductive hypothesis it previously contained $y - k$, so now it is $y - (k + 1)$), we incremented register $n + 2$ (it previously contained k , so now it is $k + 1$), and in register 0 we put $g(h(\vec{x}, k), \vec{x}, k)$, which is indeed $h(\vec{x}, k + 1)$.

So by induction, the claim holds every time we start the loop. The program halts when we start the loop with 0 in register $n + 1$. At this point we have done the loop y times (by the claim) and register 0 contains $h(\vec{x}, y)$, as desired.

The program is easily modified for the case where \vec{x} is empty. We want to use the registers as follows:

```

register  0   $h(t)$ 
register  1   $y - t$ 
register  2   $t$ 
register  3  work space
           $\vdots$ 

```

Here is the program:

I	0		
	\dots		$(m \text{ times})$
I	0		
D	1		Begin loop.
J	$\star + 3$		Halt when done.
<div style="display: flex; align-items: center; justify-content: center;"> <div style="border-left: 1px solid black; border-bottom: 1px solid black; border-right: 1px solid black; border-top: 1px solid black; width: 100px; height: 100px; margin-right: 10px;"></div> <div style="text-align: center;"> <p>Calculate g from 0, 2 to 0, preserving 3</p> </div> </div>			$t := t + 1$
I	2		Repeat loop.
J	$-(\star + 3)$		Halt.

The correctness argument continues to be applicable, with $n = 0$. ◻

Gathering together the results thus far, we have the following conclusion.

Theorem: Every primitive recursive function is register-machine computable.

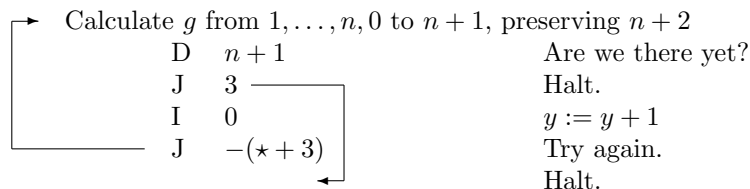
This theorem promises register-machine programs for all the primitive recursive functions from the previous chapter. But of course we can do better:

Theorem: Every general recursive partial function is register-machine computable.

Proof: We need to add the μ -operator

$$h(\vec{x}) = \mu y [g(\vec{x}, y) = 0].$$

We use the obvious program:



The program uses registers as follows:

```

register 0   y
register 1   x1
          ⋮
register n   xn
register n + 1 g( $\vec{x}$ , y)
register n + 2 work space
          ⋮

```

Of course, this program might never halt. \dashv

This theorem gives half of a significant fact: that formalizing the effective calculability concept by means of general recursiveness and formalizing the effective calculability concept by means of register machines lead to exactly the same class of partial functions. We will come to the other half soon. In particular, the theorem illustrates that register machines are capable of doing much more than might have been apparent initially from their very simple definition.

A universal program

In Chapter 1 it was argued that the “universal” partial function

$$\Phi(e, x) = \text{the result of applying the program coded by } e \text{ to input } x$$

should be a computable partial function. We now plan to verify this fact in the case of register-machine programs.

The first step is to make the phrase “program coded by e ” meaningful, by adopting a particular coding.

First, to each instruction c we will assign a number $\#c$ (called its *Gödel number*), as follows:

$$\begin{aligned}
 \#I r &= [0, r] \\
 \#D r &= [1, r] \\
 \#J q &= [2, q] \text{ if } q \geq 0 \\
 \#J q &= [3, |q|] \text{ if } q < 0
 \end{aligned}$$

Here $[x, y] = 2^{x+1} \cdot 3^{y+1}$, as in the preceding chapter. In fact, we will utilize all of the sequence-coding machinery developed there. For example, $\#I0 = [0, 0] = 6$.

And the instruction $J - 1$ has the Gödel number $[3, 1] = 144$. Observe that applying $()_0$ to the Gödel number of an instruction, we find out what type of instruction it is (increment, decrement, or jump). And by applying $()_1$ to the Gödel number of an instruction, we find the address of the register or the size of the jump.

Next, to a program c_0, \dots, c_m (i.e., a finite string of instructions), we assign its Gödel number

$$[\#c_0, \dots, \#c_m].$$

To the empty program we assign the Gödel number 1. For example, the one-line program $I0$ has the Gödel number $[6] = 2^7 = 128$. And the one-line program $J - 1$ has the Gödel number $[[3, 1]] = 2^{145}$.

For example, what is $\Phi(128, 7)$? First, we decode $128 = [6] = [\#I0]$. Next, we apply this program to 7, obtaining an output of 1. We conclude that $\Phi(128, 7) = 1$. The function Φ is not total; $\Phi(2^{25}, 7)$ is undefined. We have not yet clarified what $\Phi(e, x)$ should be when e is not the Gödel number of any program; we will take care of that matter shortly.

Digression: We could have kept these numbers smaller by using a more efficient coding technique. A program is a string of symbols over an alphabet containing the symbols $I, D, J, -$, and whatever digits we use for the numerals specifying register numbers and jump lengths. As a program, $J - 1$ is a word of length 3, in contrast to 2^{145} which, written out in base 10, takes 44 digits.

Suppose we use base-6 numerals to specify register numbers and jump lengths. For these numerals we need the digits 0, 1, 2, 3, 4, and 5. So a program can be viewed as a word over the ten-letter alphabet

$$\{I, D, J, -, 0, 1, 2, 3, 4, 5\}.$$

One very natural way to code words over this alphabet is to use “decadic notation.” That is, a word over a ten-letter alphabet *is* a numeral, and it names a number via a base-10 notation, albeit a notation somewhat different from the standard base-10 notation. For more about decadic notation, see the appendix. But for our present purposes, efficiency of coding is not required.

In a similar spirit, we can next encode the contents of all the registers into a single number. Suppose that at some instant in time, the number z_i is in register i , for each i . This information we encode into the *memory number*

$$2^{z_0} \cdot 3^{z_1} \cdot \dots \cdot p_i^{z_i} \cdot \dots = \prod_i p_i^{z_i}.$$

This “infinite product” is finite, because at any instant, only finitely many of the registers will be non-zero. (We deliberately did not use $z_i + 1$ in the exponent here.) Of course, any positive integer can be viewed as a memory number; we can take its prime factorization and read off the contents of each register. For example, a memory number of 243 tells us that register 1 contains 5 and the other registers are 0, because $243 = 3^5$.

With this coding in hand, we proceed to the second step, of constructing functions (which will turn out to be primitive recursive) for simulating the execution of a register-machine program.

One of these will be the “mem” function, which will specify how the memory number is changed by the execution of an instruction. That is, suppose that at some point the register contents are given by the memory number m and then we execute the instruction with Gödel number c . We want $\text{mem}(m, c)$ to be the new memory number, after the instruction is executed.

$$\text{mem}(m, c) = \begin{cases} m \cdot p_{(c)_1} & \text{if } (c)_0 = 0 \text{ and } c \neq 0 & \text{(increment)} \\ \lfloor m/p_{(c)_1} \rfloor & \text{if } (c)_0 = 1 \text{ and } p_{(c)_1} \mid m & \text{(decrement)} \\ m & \text{otherwise.} \end{cases}$$

For example, $\text{mem}(m, 6) = 2m$ for any number m . This is because 6 is the Gödel number of the instruction I0, and executing this instruction increments the exponent of 2 in m 's prime factorization. What is $\text{mem}(m, 12)$? 12 is the Gödel number of the instruction D0. If m is even, then $\text{mem}(m, 12) = m/2$, which decrements the exponent of 2 in m 's prime factorization. But if m is odd, then $\text{mem}(m, 12) = m$.

Observe that this equation for the mem function is expressed entirely within the language we have built up for primitive recursiveness. That is, simply from the *form* of the equation, we see that mem is a primitive recursive function. It is built up using definition by cases, the prime-counting function p_n, \dots . And because we know that all primitive recursive functions are register-machine computable, we know we can make a register-machine program that computes mem.

Next suppose we have a program with Gödel number

$$e = [\#c_0, \dots, \#c_m].$$

So $\text{lh } e = m + 1$, the number of instructions in the program. We want to think about the *location counter*, which keeps track of where we are in the program. If the location counter is 0, then we are about to execute c_0 , the first instruction. More generally, if the location counter is k , then we are about to execute c_k , the $(k + 1)$ st instruction. If the location counter is $\text{lh } e$, then the program has come to a good halt, by seeking the first non-existent instruction. (A location counter greater than $\text{lh } e$ would correspond to a bad halt, seeking a non-existent instruction later than the first such.)

We want to define a function “loc” that gives the value of the location counter. That is, if the location counter is now k , then we want $\text{loc}(k, m, e)$ to be the new value of the location counter after we execute c_k , the $(k + 1)$ st instruction, where m is the memory number.

What does this function need to do? First of all, it needs to find $(e)_k = \#c_k$, the Gödel number of c_k . This number is a pair

$$(e)_k = [((e)_k)_0, ((e)_k)_1]$$

where $((e)_k)_0$ is 0, 1, 2, or 3. If $((e)_k)_0 = 1$, then we have a decrement instruction, and it will be necessary to check the memory number to see if the decrement is successful or not. Here is the function written out in full:

$$\text{loc}(k, m, e) = \begin{cases} k & \text{if } k \geq \text{lh } e & \text{(already halted)} \\ k + 2 & \text{if } ((e)_k)_0 = 1 \text{ and} \\ & p_{((e)_k)_1} \mid m \text{ and} & \text{(decrement)} \\ & k + 2 \leq \text{lh } e \\ \min(k + ((e)_k)_1, \text{lh } e) & \text{if } ((e)_k)_0 = 2 & \text{(jump forward)} \\ k \dot{-} ((e)_k)_1 & \text{if } ((e)_k)_0 = 3 \text{ and} \\ & ((e)_k)_1 \leq k & \text{(jump back)} \\ \text{lh } e & \text{if } ((e)_k)_0 = 3 \text{ and} \\ & ((e)_k)_1 > k & \text{(bad jump)} \\ k + 1 & \text{otherwise} & \text{(default)} \end{cases}$$

For example, what is $\text{loc}(0, m, 128)$? Here $k = 0$, so we are at the very beginning of the program. And $e = 128$, which we need to decode. Since $e = 128 = 2^7$, we see that we have a one-line program (that is, $\text{lh}(e) = 1$). The one instruction has Gödel number $(e)_0 = 6$. We see that $((e)_0)_0 = 0$, so we have an increment instruction. And $((e)_0)_1 = 0$, so the register in question is register 0. (In other words, $6 = \#D0$.) In the above equation, it is the last line (the “default” line) that applies: $\text{loc}(0, m, 128) = k + 1 = 0 + 1 = 1$. The new value of the location counter is 1. This is just as it should be; having executed the I0 instruction, we push the location counter up one.

And next we come to $\text{loc}(1, 2m, 128)$. But the program has already halted. In the above equation, it is now the first line that is applicable. That is, because $1 = k \geq \text{lh } e = 1$, we obtain $\text{loc}(1, 2m, 128) = 1$.

The equation may be a bit unwieldy, but each piece of the equation does the natural thing: skip, jump forward, jump back, or go on to the next instruction, as the case may be. (If the program in question is trying to make a bad jump, either to a point before the beginning of the program or to a point later than the first non-existent instruction, then the loc function generously tries to make the situation appear to be a little better than it really is.)

Again, from looking at the form of the equation, we realize that the loc function is primitive recursive. And being primitive recursive, it is therefore computable by a register program. (In particular, loc is a total function. Even if e is *not* the Gödel number of a program at all, $\text{loc}(k, m, e)$ will be defined. That is, if we put garbage into the loc function, then we get garbage out, but at least we get something out. And it is not hard to see that we always have $\text{loc}(k, m, e) \leq \max(k, \text{lh } e)$.)

Now we are ready to describe our universal program (that computes Φ). In computing $\Phi(e, x)$, initially e is in register 1 and x is in register 2, and the other registers are blank. The program will keep the location counter k in register 3 (initially 0), the memory number m in register 4, and the Gödel number $(e)_k$ of the next instruction in register 5.

register	0	number of remaining instructions
register	1	program e
register	2	input x
register	3	location k
register	4	memory m
register	5	instruction $(e)_k$
register	6	work space
	\vdots	

Here is our universal program, described in nine or ten lines:

	Calculate 3^x from 2 to 4, preserving 4	Initialize memory.
→	Calculate $(lh\ e) \div k$ from 1, 3 to 0, preserving 5	Start loop here.
	D 0	Done?
	J *	Exit loop.
	Calculate $(e)_k$ from 1, 3 to 5, preserving 5	Get command.
	Calculate loc from 3, 4, 1 to 3, preserving 6	Update location.
	Calculate mem from 4, 5 to 4, preserving 6	Update memory
	J - **	Start loop again.
→	Calculate $(m)_0^*$ from 4 to 0, preserving 0	Extract output.
		Halt.

(Here $(m)_0^* = \mu t < m[2^{t+1} \uparrow m]$, the exponent of 2 in the prime factorization of m .) This program, given x and e , decodes e to see what it says to do with x , and then does it (to paraphrase what was said on page 107).

First consider the case where e is the Gödel number of a program \mathcal{P} that computes a 1-place partial function f . Then the universal program will mimic the operation of \mathcal{P} on input x . If $f(x)$ is defined, then the universal program will halt with $f(x)$ in register 0; if $f(x)$ is undefined, then the universal program will not halt. (What we have done is to make explicit just what “executing” a program \mathcal{P} involves.)

Secondly, consider what happens if e is the Gödel number of a program \mathcal{P} that does not compute any partial function. This can happen if \mathcal{P} has some bad jumps (i.e., a jump to a non-existent instruction other than the first such one). Again, the universal program will mimic the operation of \mathcal{P} on input x . If a bad jump is encountered, the universal program will halt (by setting the location counter to $lh\ e$). But the universal program will come to a good halt (with output $(m)_0^*$, where m is the memory number at the time).

Thirdly, it may be that e is not the Gödel number of a program at all. The universal program will nonetheless start running. After all, both *mem* and *loc* are *total* functions. Maybe at the conclusion of some loop, the location counter will happen to equal $lh\ e$. Then the universal program will halt (by seeking the first non-existent instruction). And maybe that will never happen.

The point is that the universal program computes some partial 2-place function Φ . Here $\Phi(e, x)$ is whatever the universal program gives us, if and when it halts on input e and x .

Because Φ is a register-machine computable partial function, it follows that for any fixed e , the 1-place function $x \mapsto \Phi(e, x)$ is register-machine computable (one program for it puts the constant e into register 2 and then runs the universal program). Call this function $\llbracket e \rrbracket$:

$$\llbracket e \rrbracket(x) = \Phi(e, x)$$

That is, $\llbracket e \rrbracket(x)$ is whatever this universal program produces (if anything), given the input $\langle e, x \rangle$.

Then if e is the Gödel number of a program that computes a 1-place partial function f , we can conclude that $\llbracket e \rrbracket = f$. And if e is some other number (either the number of a program that does not compute a partial function, or perhaps not the Gödel number of any program at all), then we can say at least that $\llbracket e \rrbracket$ is *some* register-machine computable partial function.

Whenever e is a number for which $\llbracket e \rrbracket$ is the function f , we will say that e is an *index* of f . Thus the indices of computable partial function f include the Gödel numbers of programs that compute f , and also include any other numbers for which $\llbracket e \rrbracket$ just happens to be the function f (that is, $\Phi(e, x) = f(x)$ for all x).

In summary, we now have the following result.

Enumeration theorem:

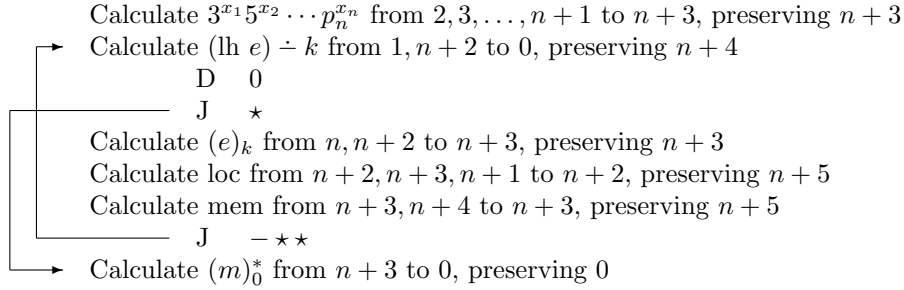
- (i) Φ is a register-machine computable 2-place partial function.
- (ii) For each number e , $\llbracket e \rrbracket$ is a register-machine computable 1-place partial function.
- (iii) Each 1-place register-machine computable partial function is $\llbracket e \rrbracket$ for some number e .

Thus

$$\llbracket 0 \rrbracket, \llbracket 1 \rrbracket, \llbracket 2 \rrbracket, \dots$$

is a complete list (with repetitions) of all the 1-place register-machine computable partial functions, and only those.

It is straightforward to generalize these ideas from 1-place partial functions to n -place partial functions. Our universal programs starts with e, x_1, x_2, \dots, x_n in registers 1, 2, \dots , $n, n+1$, and with 0 in the other registers. The program will keep the location counter k in register $n+2$, the memory number m in register $n+3$, and the Gödel number $(e)_k$ of the next instruction is register $n+4$. Here it is:



This program computes an $(n+1)$ -place partial function $\Phi^{(n)}$. For each fixed e , define the n -place partial function $\llbracket e \rrbracket^{(n)}$ by the equation

$$\llbracket e \rrbracket^{(n)}(x_1, x_2, \dots, x_n) = \Phi^{(n)}(e, x_1, x_2, \dots, x_n).$$

Enumeration theorem:

- (i) $\Phi^{(n)}$ is a register-machine computable $(n+1)$ -place partial function.
- (ii) For each number e , $\llbracket e \rrbracket^{(n)}$ is a register-machine computable n -place partial function.
- (iii) Each n -place register-machine computable partial function is $\llbracket e \rrbracket^{(n)}$ for some number e .

Thus

$$\llbracket 0 \rrbracket^{(n)}, \llbracket 1 \rrbracket^{(n)}, \llbracket 2 \rrbracket^{(n)}, \dots$$

is a complete list (with repetitions) of all the n -place register-machine computable partial functions, and only those.

One significant benefit of having universal functions is that we can apply diagonalization to them. The diagonal function

$$d(x) = \llbracket x \rrbracket(x) + 1$$

is a register-machine computable partial function (being obtained from Φ , I_1^1 , and S by composition). And so $d = \llbracket e \rrbracket$ for some number e . What can we say about $d(e)$? We have the equation

$$d(e) = \llbracket e \rrbracket(e) + 1 = d(e) + 1$$

but ‘=’ means that either both sides are undefined, or both sides are defined and are equal. We can conclude the $d(e)$ must be undefined, lest $0 = 1$.

(Another diagonal function

$$\hat{d}(x) = 1 \div \llbracket x \rrbracket(x)$$

would serve just as well here. And \hat{d} has the added advantage of being bounded by 1.)

By contrast, suppose we attempt to change d into a total function:

$$D(x) = \begin{cases} \llbracket x \rrbracket(x) + 1 & \text{if this is defined} \\ 0 & \text{otherwise} \end{cases}$$

Then D is *not* register-machine computable. We cannot possibly have $D = \llbracket e \rrbracket$ because either $\llbracket e \rrbracket(e)$ is defined and $D(e)$ is larger by 1, or else $\llbracket e \rrbracket(e)$ is undefined and $D(e) = 0$. In fact, the same argument yields a slightly stronger statement:

Theorem: (a) There is no total register-machine computable function that extends the diagonal function $d(x) = \llbracket x \rrbracket(x) + 1$.

(b) There is no total register-machine computable function that extends the diagonal function $\hat{d}(x) = 1 \div \llbracket x \rrbracket(x)$.

Let K be the domain of the diagonal function:

$$K = \{x \mid \llbracket x \rrbracket(x) \downarrow\}$$

The semi-characteristic function of K

$$c_K(x) = \begin{cases} 1 & \text{if } x \in K \\ \uparrow & \text{if } x \notin K \end{cases}$$

is a register-machine computable partial function. To compute $c_K(x)$, we first try to compute $\llbracket x \rrbracket(x)$; if and when we succeed we give output 1. Or in equation form, $c_K(x) = 1 + 0 \cdot \llbracket x \rrbracket(x)$.

But now consider the full characteristic function of K :

$$C_K(x) = \begin{cases} 1 & \text{if } x \in K \\ 0 & \text{if } x \notin K \end{cases}$$

Theorem: C_K is *not* register-machine computable.

Proof: Suppose that, to the contrary, C_K was register-machine computable. Then the above diagonal function D would be computed by the following program:

Calculate $C_K(x)$ from 1 to 0, preserving 2	Decide if $x \in K$.
D 0	Yes?
J $\star + 1$	Halt.
Calaculate $d(x)$ from 1 to 0, preserving 2	Halt.

where \star is the length of the program used for d . \dashv

Unsolvability of the halting problem: The total function

$$H(x, y) = \begin{cases} 1 & \text{if } \llbracket x \rrbracket(y) \downarrow \\ 0 & \text{if } \llbracket x \rrbracket(y) \uparrow \end{cases}$$

is *not* register-machine computable.

Proof: $C_K(x) = H(x, x)$. \dashv

In terms of binary relations, this result says that the characteristic function of

$$\{(x, y) \mid \llbracket x \rrbracket(y) \downarrow\}$$

(which is the domain of Φ) is not register-machine computable. Its semi-characteristic function *is* register-machine computable, as in the case of K .

What we are doing here is retracing some of the material that appeared in Chapter 1 in informal terms, but now formalized by using register machines as our model of computability, and by exploiting the development in Chapter 2 of general recursive functions.

We can extract even more from these ideas, if we add a “clock.” That is, instead of looking at $\Phi(e, x)$, we will add a third variable for time t , and determine, for a triple $\langle e, x, t \rangle$, where the calculation of the program with Gödel number e and input x stands after t steps. More specifically, we want to determine both the location counter and the memory number after t steps.

The pair

$$[\text{location counter, memory number}]$$

gives us a “snapshot” showing the status of the calculation. So what we want is a “snap” function such that $\text{snap}(e, x, t)$ gives the snapshot after t steps.

For a start, what is $\text{snap}(e, x, 0)$? No steps have been executed, so nothing has happened yet. The location counter is 0 and the memory number is 3^x :

$$\text{snap}(e, x, 0) = [0, 3^x]$$

which happens to be $2 \cdot 3^{(3^x+1)}$. For example, a snapshot of 162 tells us that the location counter is 0, register 1 contains 1, and the other registers are 0. (This holds because $162 = 2 \cdot 81 = 2 \cdot 3^{3+1} = [0, 3]$.)

Now suppose we know, for some number t of steps, the snapshot

$$\text{snap}(e, x, t) = [k, m].$$

What comes next? The next instruction to execute is $(e)_k$, if $k < \text{lh } e$. The memory number will change from m to $\text{mem}(m, (e)_k)$. The location counter will change from k to $\text{loc}(k, m, e)$. Putting these pieces together gives us the equation:

$$\text{snap}(e, x, t+1) = [\text{loc}(k, m, e), \text{mem}(m, (e)_k)]$$

Or noting that $k = (\text{snap}(e, x, t))_0$ and $m = (\text{snap}(e, x, t))_1$ yields the equation:

$$\begin{aligned} \text{snap}(e, x, t+1) = \\ [\text{loc}((\text{snap}(e, x, t))_0, (\text{snap}(e, x, t))_1, e), \text{mem}((\text{snap}(e, x, t))_1, (e)_{(\text{snap}(e, x, t))_0})] \end{aligned}$$

So we now have a pair of recursion equations for snap :

$$\begin{aligned} \text{snap}(e, x, 0) &= [0, 3^x] \\ \text{snap}(e, x, t+1) &= \\ &[\text{loc}((\text{snap}(e, x, t))_0, (\text{snap}(e, x, t))_1, e), \text{mem}((\text{snap}(e, x, t))_1, (e)_{(\text{snap}(e, x, t))_0})]. \end{aligned}$$

Moreover, the recursion equations use only known primitive recursive pieces. We conclude that the snap function is primitive recursive (and hence register-machine computable).

Observe that snap is a *total* function. Even if e is the Gödel number of a weird program, or even if e is not the Gödel number of a program at all, the quantity $\text{snap}(e, x, t)$ is defined for all x and t . Compare this with the universal program for $\Phi(e, x)$. In both cases, we are starting with a memory number of 3^x and a location counter of 0, and then applying the functions mem and loc over and over. But for $\text{snap}(e, x, t)$, we get to stop after t steps.

Of course, we are particularly interested in the case where e is indeed the Gödel number of a program \mathcal{P} that computes a partial function f . In this case, whenever $x \in \text{dom } f$, then sooner or later we will reach a snapshot where the location counter says the computation has halted:

$$(\text{snap}(e, x, t))_0 \geq \text{lh } e$$

We might think of this situation as a cause for celebration. Accordingly, we define the ternary relation T

$$T = \{\langle e, x, t \rangle \mid (\text{snap}(e, x, t))_0 \geq \text{lh } e\}$$

or in other words

$$\begin{aligned} T(e, x, t) &\iff (\text{snap}(e, x, t))_0 \geq \text{lh } e \\ &\iff \llbracket e \rrbracket(x) \downarrow \text{ in } \leq t \text{ steps.} \end{aligned}$$

We observe that the relation T is primitive recursive. And the partial function

$$\begin{aligned} \langle e, x \rangle &\mapsto \mu t T(e, x, t) \\ &\mapsto \mu t (\llbracket e \rrbracket(x) \downarrow \text{ in } \leq t \text{ steps}) \end{aligned}$$

measures the running time of e at x (where the running time is undefined when the computation goes on forever).

In general, we cannot put an upper bound on the μ -operator here. And by the unsolvability of the halting problem, we do not in general know when the search for t will succeed and when it will go on forever. Nonetheless, we can at least assert that the running-time function is a *general* recursive *partial* function, because it is obtained by applying search to a primitive recursive relation. (It is certainly not a total function.)

The “terminal snapshot” is $\text{snap}(e, x, \mu t T(e, x, t))$, if this is defined at all. (We can think of this as $\lim_{t \rightarrow \infty} \text{snap}(e, x, t)$, a limit that might or might not exist. Once we reach the terminal snapshot, if we do, then the snap function stops changing—the functions mem and loc have been constructed in such a way as to make sure of this.) From the terminal snapshot, we can apply $()_1$ to obtain the terminal memory number. And then we can apply to that quantity the function $()_0^*$ to obtain the contents of register 0 at termination. And that is the output of the calculation. Thus we obtain the following conclusion:

Normal form theorem: For any x and e ,

$$\begin{aligned} \llbracket e \rrbracket(x) &= \Phi(e, x) = ((\text{snap}(e, x, \mu t T(e, x, t)))_1)_0^* \\ &= ((\text{snap}(e, x, \mu t[(\text{snap}(e, x, t))_0 \geq \text{lh } e]))_1)_0^* \end{aligned}$$

where as usual ‘=’ means that either both sides are undefined or else both sides are defined and equal.

In other words, we can break down the calculation of $\llbracket e \rrbracket(x)$ into four steps:

- Find $t_{\text{halt}} = \mu t T(e, x, t)$.
- Find the terminal snapshot $\text{snap}_{\text{halt}} = \text{snap}(e, x, t_{\text{halt}})$.
- Find the memory number $m_{\text{halt}} = (\text{snap}_{\text{halt}})_1$.
- Find the output value $(m_{\text{halt}})_0^*$.

The first step uses a general recursive function; the other steps use primitive recursive functions.

It is straightforward to extend these ideas to functions of more than one variable: $\text{snap}^{(2)}(e, x_1, x_2, t)$ is obtained by starting with

$$\text{snap}^{(2)}(e, x_1, x_2, 0) = [0, 3^{x_1} \cdot 5^{x_2}]$$

and proceeding as before. We define the $(n+2)$ -ary relation $T^{(n)}$

$$\begin{aligned} T^{(n)}(e, \vec{x}, t) &\iff (\text{snap}^{(n)}(e, \vec{x}, t))_0 \geq \text{lh } e \\ &\iff \llbracket e \rrbracket^{(n)}(\vec{x}) \downarrow \text{ in } \leq t \text{ steps.} \end{aligned}$$

Again we observe that the relation $T^{(n)}$ is primitive recursive.

Normal form theorem: For any n , e , and \vec{x} ,

$$\begin{aligned} \llbracket e \rrbracket^{(n)}(\vec{x}) &= \Phi^{(n)}(e, \vec{x}) = ((\text{snap}^{(n)}(e, \vec{x}, \mu t T^{(n)}(e, \vec{x}, t)))_1)_0^* \\ &= ((\text{snap}^{(n)}(\vec{x}, e, \mu t[(\text{snap}^{(n)}(e, \vec{x}, t))_0 \geq \text{lh } e]))_1)_0^*. \end{aligned}$$

Looking at the right-hand side in this equation, we observe that it defines a general recursive partial function. In fact everything on the right side is primitive recursive, except for the single application of the μ -operator. Hence we have the following:

Corollary: Every register-machine computable partial function is general recursive.

Putting this corollary together with an earlier theorem, we conclude that the class of register-machine computable partial functions is exactly the same as the class of general recursive functions. This result is a welcome byproduct of our analysis of register-machine computations.

The methods used here to obtain the equivalence of general recursiveness to register-machine computability are adaptable to obtaining equivalence between

other formalizations of effective calculability that were described in Chapter 1. For example, to show that Turing machines compute *at least* as many functions as the other approaches give us, it suffices to show first that Turing machines can compute the zero, successor, and projection functions, and secondly that the class of Turing-computable partial functions is closed under composition, primitive recursion, and search. In the other direction, to show that Turing machines compute *at most* the functions given by the other approaches, we can code the Turing machines in a suitable way, construct a universal Turing machine, and prove a normal form theorem. There are textbooks that do exactly that.

As another consequence of the normal form theorem, we can represent the domain of the partial function $\llbracket e \rrbracket$ in the following form:

$$\begin{aligned} x \in \text{dom } \llbracket e \rrbracket &\iff \exists t[(\text{snap}(e, x, t))_0 \geq \text{lh } e] \\ &\iff \exists t T(e, x, t) \\ \vec{x} \in \text{dom } \llbracket e \rrbracket^{(n)} &\iff \exists t[(\text{snap}^{(n)}(e, \vec{x}, t))_0 \geq \text{lh } e] \\ &\iff \exists t T^{(n)}(e, \vec{x}, t) \end{aligned}$$

On the right, we have a primitive recursive relation, prefixed by an (unbounded) “ $\exists t$ ” quantifier.

Digression: The relation T (or more generally, $T^{(n)}$) that we have constructed is closely related to what is generally called “the Kleene T -predicate.” There are technical differences, however.

Exercises

1. Give a register-machine program that computes the function

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

2. (a) Show that the set of Gödel numbers of instructions is a primitive recursive set.

- (b) Show that the set of Gödel numbers of programs is a primitive recursive set.

3. Determine $\llbracket 0 \rrbracket(x)$ for all values of x for which this is defined. (Note that 0 is not the Gödel number of any program.)

4. We know that the one-place function that is constantly equal to k is computable. Show that there is a primitive recursive function f such that $\llbracket f(k) \rrbracket$ is that function. That is, we need the equation $\llbracket f(k) \rrbracket(x) = k$ to hold for all k and x .

5. Show that the partial function

$$\text{time}(e, x) = \mu t T(e, x, t)$$

is not bounded by any total computable function. That is, show that there is no total computable function F with the property that

$$\text{time}(e, x) \leq F(e, x)$$

whenever the left side is defined.

6. Assume that h is a primitive recursive function and e is the Gödel number of a program such that

$$\llbracket e \rrbracket(x) \downarrow \text{ in } \leq h(x) \text{ steps}$$

for all x . Show that the function $\llbracket e \rrbracket$ is primitive recursive. (That is, a program that runs in primitive recursive time always computes a primitive recursive function.)

7. Explain why, for a calculation that eventually halts, all of the snapshots that arise in the course of the calculation must be distinct.

Register machines over words¹

The inputs to effective procedures are not really numbers, but numerals—strings of symbols. For example, the input to a Turing machine consists of a string of symbols written on consecutive squares of its tape. The register machines we have been considering up to now can be thought of as working with “base-1” numerals, where the numeral for 7 is the string

||||||

of seven tally marks. In base-1 notation, the increment and decrement commands are the natural ones to use.

But suppose that we wanted our machines to work with binary numerals. In this case, each register would contain some string (possibly empty) of 0’s and 1’s. Now the increment command seems less natural; changing 111 to 1000 is a big change, in some sense—every symbol changed.

We want to extend the register machine concept to the situation where each register contains a string (possibly empty) of symbols from a q -letter alphabet

$$\Sigma = \{a_1, \dots, a_q\}.$$

Actually, the alphabet is an *ordered* set,

$$\Sigma = \langle a_1, \dots, a_q \rangle$$

because alphabetical order will matter.

For the machines we have been considering up to now, $q = 1$ and $\Sigma = \langle | \rangle$. For binary notation, we would use $q = 2$ and $\Sigma = \langle 0, 1 \rangle$.

To simplify the exposition, we will fix a particular size of alphabet, namely $q = 3$, where $\Sigma = \langle a, b, c \rangle$. But it will be clear how to adjust the concepts to larger or smaller values of q .

The commands, as before, are of three types: increment, decrement, and jump:

- $I_a r$ (where r is a numeral for a natural number). “Increment register r by a .” The effect of this instruction is to prefix the letter a to the (left) end of the word in register r . If the register was previously empty, then it will now contain the one-letter word a . The machine then proceeds to the next instruction in the program (if any).
- $I_b r$ (where r is a numeral for a natural number). “Increment register r by b .” The effect of this instruction is the same, but it prefixes the letter b to the (left) end of the word in register r .
- $I_c r$ (where r is a numeral for a natural number). “Increment register r by c .” This instruction prefixes the letter c .

¹This material will be needed in Chapter 7.

- $D\ r$ (where r is a numeral for a natural number). “Decrement register r .” The effect of this instruction depends on the contents of register r . If the word in register r is empty, the machine simply proceeds to the next instruction, without changing the contents of the register. But if the word is non-empty, then the last (rightmost) letter is deleted. And what the machine does next depends on that deleted letter.
 - If the deleted letter was a , then the machine skips one instruction, and goes to the next one after that (if any).
 - If the deleted letter was b , then the machine skips two instructions, and goes to the next one after that (if any).
 - If the deleted letter was c , then the machine skips three instructions, and goes to the next one after that (if any).

In summary, the machine tries to delete the last letter in register r and if it is successful then it skips the appropriate number of instructions.

- $J\ n$ (where n is a numeral for an integer in \mathbb{Z}). “Jump n .” All registers are left unchanged. The machine takes as its next instruction the n th instruction following this one in the program (if $n \geq 0$), or the $|n|$ th instruction preceding this one (if $n < 0$). The machine halts if there is no such instruction in the program. An instruction of $J\ 0$ results in a loop, with the machine executing this one instruction over and over again.

Observe that in the case of a *one*-letter alphabet, the preceding list of commands is the same list we have been considering before.

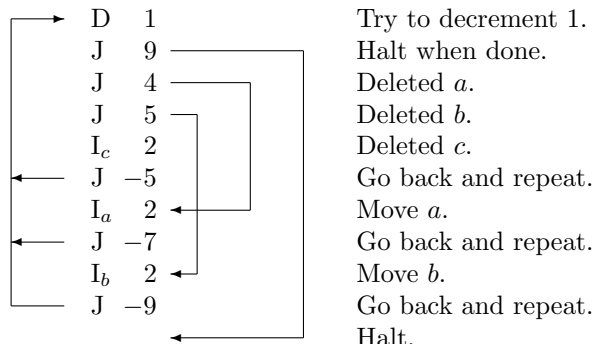
Example: The five-line program (call it CLEAR 3)

→	D	3	Delete last letter.
	J	4	Halt when done.
←	J	−2	Go back and repeat.
←	J	−3	Go back and repeat.
←	J	−4	Go back and repeat.
			Halt.

will clear whatever is in register 3.

Example: The following program will take the word in register 1 and prefix it to whatever was in register 2. So if register 2 was empty, this program will simply move the word from register 1 to register 2. In any case, the concatenation of the words in registers 1 and 2 will be placed in register 2. At the end,

register 1 will be empty.



This program consists of ten instructions. Call it PREFIX 1 to 2.

We can use this program to do a “right increment,” that is, to *append* a letter to a given word. Assume we know that register 7 is empty. Then the following program will append the letter *b* to the right end of whatever word is in register 0:

```

                Ib 7
    PREFIX 0 to 7
    PREFIX 7 to 0
    
```

Call this APPEND *b* to 0. It consists of 21 instructions. In using this program, we need to know of an empty register (in this example, register 7).

Now suppose we think of the words over our alphabet as being *numerals*, that is, as naming numbers. Because we have a three-letter alphabet, we treat words as base-3 numerals. More specifically, we use *triadic* notation, where the letters *a*, *b*, and *c* name 1, 2, and 3 respectively:

$$v(a) = 1, \quad v(b) = 2, \quad v(c) = 3$$

Then the general rule is that a *k*-letter word

$$s_k s_{k-1} \cdots s_1$$

names $v(s_k)3^{k-1} + v(s_{k-1})3^{k-2} + \cdots + v(s_2)3 + v(s_1)$, and the empty word λ names 0. Then we obtain a one-to-one correspondence between the set of all words and the set of natural numbers. (See the appendix for a discussion of these numerals.) Here is a list of the first few numerals:

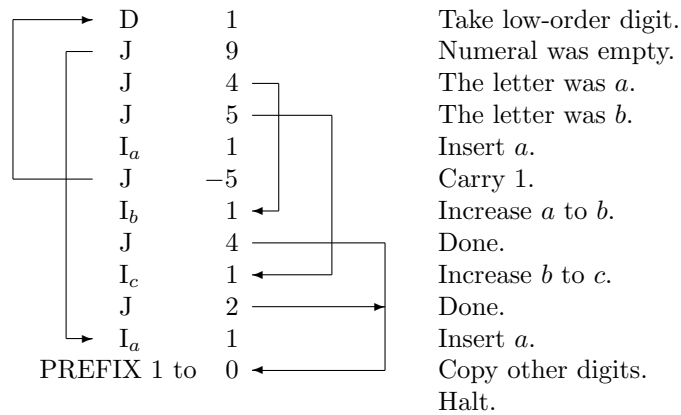
numeral	number
λ	0
a	1
b	2
c	3
aa	4
ab	5
ac	6
ba	7
bb	8
bc	9
ca	10
cb	11
cc	12
aaa	13
aab	14
...	...

How do we add one in triadic notation? We look at the rightmost letter in the numeral. If that letter is a or b , then we simply increase it to the next letter. But if the rightmost letter is c , then we replace it by a (this lowers the number by 2), and we carry 1 to the left (which raises the number by 3). Here are some examples:

$$bca + a = bcb, \quad ccc + a = aaaa, \quad acc + a = baa$$

Let's make a register-machine program to do this. That is, we want a program that computes the successor function, in triadic notation.

Assume the given word (the given numeral) is in register 1, and that register 0 is initially empty.



This program leaves the output in register 0, with register 1 empty. Call it ADD1 from 1 to 0.

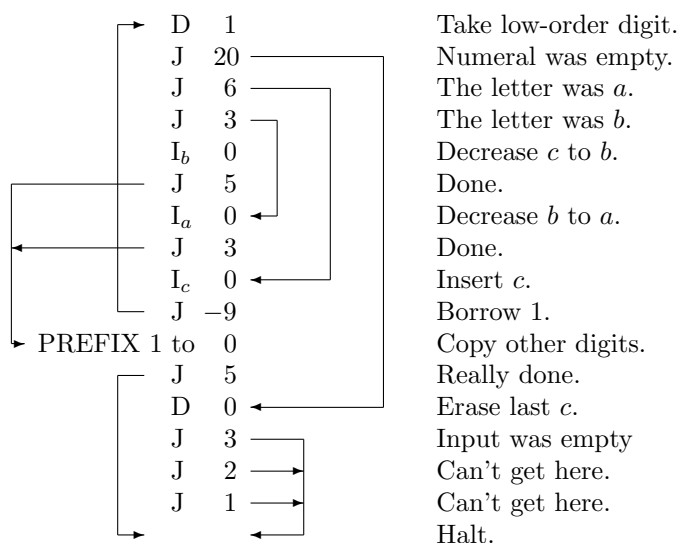
Subtracting one (that is, computing the predecessor function) is very similar, except that instead of carrying to the left, we borrow from the left. And of course, we cannot subtract from λ , the numeral for 0.

Here are some examples:

$$bcb - a = bca, \quad baa - a = acc, \quad aaaa - a = ccc$$

In general, we look at the rightmost digit. If it is b or c , then we simply lower it to a or b , respectively, and we are done. But if it is an a , then we replace it by c and borrow one from the left.

Assume the given word (the given numeral) is in register 1, and that register 0 is initially empty.



This program leaves the output in register 0, with register 1 empty. Call it SUB1 from 1 to 0. If applied to the empty word, the output is also empty.

Theorem: Every n -place general recursive partial function f is register-machine computable in the following sense: There is a program \mathcal{P} such that if we start a register machine with the *triadic numerals* for x_1, \dots, x_n in registers $1, \dots, n$ and λ in the other registers and we apply program \mathcal{P} , then the following conditions hold:

- If $f(x_1, \dots, x_n)$ is defined, then the computation eventually terminates with the *triadic numeral* for $f(x_1, \dots, x_n)$ in register 0. Furthermore, the computation terminates by seeking a $(p + 1)$ st instruction, where p is the length of \mathcal{P} .
- If $f(x_1, \dots, x_n)$ is undefined, then the computation never terminates.

The proof is much as in the base-1 case, *mutatis mutandis*. The zero functions are computed by the empty program, and by many others. The successor

function is computed by ADD1 from 1 to 0. The projection function I_n^k is computed by PREFIX n to 0.

Closure under composition is a matter of good organization and careful bookkeeping. For closure under primitive recursion, we use both the SUB1 program and the ADD1 program. Closure under the μ -operator uses the ADD1 program.

Tools for writing and evaluating register-machine programs over a two-letter alphabet have been made available on the web by Lawrence Moss; see <http://www.indiana.edu/~iulg/trm>. The alphabet used there is $\{1, \#\}$.

Exercises

8. Modify the program PREFIX 1 to 2 for a two-letter alphabet $\Sigma = \langle a, b \rangle$.
9. Give a program that takes the first (i.e., leftmost) symbol (if any) from the word in register 2, and puts it into register 1 (assumed to be initially empty). At the end, register 2 should be empty.
10. Give a program (call it REVERSE 1 to 0) that takes the input word in register 1 and copies it into register 0, backwards. That is, the letters in the output word must be the same as the letters in the input word, but in the opposite order.
11. Modify the program REVERSE 1 to 0 from the previous exercise for a two-letter alphabet $\Sigma = \langle a, b \rangle$.
12. Give a program that takes the last (i.e., rightmost) letter from the word in register 1, and prefixes it to the left of the word in register 0. But at the end, the word in register 1 is to be unchanged.
13. Modify the program ADD1 from 1 to 0 for a two-letter alphabet $\Sigma = \langle a, b \rangle$.
14. Modify the program SUB1 from 1 to 0 for a two-letter alphabet $\Sigma = \langle a, b \rangle$.

Binary arithmetic

Addition and multiplication are primitive recursive functions. So by a recent theorem, there are programs to compute them (in triadic notation). But the proof to that theorem yields very slow programs. The program for addition would calculate $x + y$ by going through a loop y times, adding 1 to x each time it executes the loop. If y is a huge number, this is going to take a huge amount of time; it is going to take an amount of time proportional to y .

The way we learned to add in second grade is a great deal faster. We start by adding the low-order digits of x and y . This gives us the low-order digit of the sum, and tells us whether or not we need to carry a digit. Then we keep going until we are done. The amount of time will be roughly proportional to the *length* of the numeral for y .

Let's look at this a little more carefully. But changing the scene, we take the two-letter alphabet $\Sigma = \langle 0, 1 \rangle$ and we use standard binary numerals (not dyadic numerals). So for addition, we need a binary adder. If y is a 100-bit number (i.e., $|y| = 100$, where $|y|$ is the number of bits in the binary numeral for the number y), we expect to go through the program's loop 100 times. But a 100-bit number is at the very least 2^{99} , and we definitely do not want to go through the program's loop 2^{99} times.

In outline, we know how a binary adder works. Initially, x is in register 1 (in binary), y is in register 2 (in binary), and the other registers contain the empty word λ . The $(k + 1)$ st time through the main loop (initially $k = 0$), register 0 will contain the k low-order bits of the sum, registers 1 and 2 will contain x and y except for their k low-order bits, and register 3 will contain either 1 (to show a carry bit) or λ (to show there is no carry bit).

How long will this program take? Each time through the loop, we shorten the words in registers 1 and 2 by one bit. So the number of times we go through the loop is bounded by $\min(|x|, |y|)$, the length of the shorter input numeral. Once we exit the main loop, the leftover bits from the longer input numeral need to be prefixed to the sum numeral. So the number of steps to obtain the sum will be

$$\text{constant} \times \max(|x|, |y|)$$

where the constant depends of the program, but will be less than the length of the program.

Now what about multiplication? Again, it would be much too slow to compute xy by adding x to itself y times. In the third grade, we all learned a much faster algorithm. This algorithm involves going through a certain loop a number of times equal to $|y|$, the *length* of y . Each time through the loop, we either do nothing (if the bit in y is 0), or we add x —suitably shifted—to the sum being accumulated (if the bit in y is 1).

We could code this algorithm into a suitable program. But instead, let's leap to the real point: The program will produce xy (in binary) in a number of steps that is bounded by

$$\text{constant} \times |y| \times \max(|x|, |y|)$$

for some constant.