Computability Theory

This section is partly inspired by the material in "A Course in Mathematical Logic" by Bell and Machover, Chap 6, sections 1-10.
Other references: "Introduction to the theory of computation" by Michael Sipser, and "Computability, Complexity, and Languages" by M. Davis and E. Weyuker.

Our first goal is to give a formal definition for what it means for a function on $\mathbb{N}$ to be computable by an algorithm. Historically the first convincing such definition was given by Alan Turing in 1936, in his paper which introduced what we now call Turing machines. Slightly before Turing, Alonzo Church gave a definition based on his lambda calculus. About the same time Gödel, Herbrand, and Kleene developed definitions based on recursion schemes. Fortunately all of these definitions are equivalent, and each of many other definitions proposed later are also equivalent to Turing's definition. This has lead to the general belief that these definitions have got it right, and this assertion is roughly what we now call "Church's Thesis".

Our first definition will be based on a simple computer model called Register Machines, something proposed by Shepherdson and Sturgis in the 1960's. Then we will give a recursion-theoretic definition due to Kleene, and prove our two definitions are equivalent.

A natural definition of computable function f on $\mathbb{N}$ allows for the possibility that $f(x)$ may not be defined for all $x \in \mathbb{N}$, because algorithms do not always halt. Thus we will use the symbol ∞ to mean "undefined".

Definition: A *partial function* is a function

$$f : (\mathbb{N} \cup \{\infty\})^n \to \mathbb{N} \cup \{\infty\}, n \geq 0$$

such that $f(c_1, ..., c_n) = \infty$ if some $c_i = \infty$.

In the context of computability theory, whenever we refer to a function on $\mathbb{N}$, we mean a partial function in the above sense.

Definitions:

$$\text{Domain}(f) = \{\vec{x} \in \mathbb{N}^n \mid f(\vec{x}) \neq \infty\}$$

where $\vec{x} = (x_1 \cdots x_n)$ We say f is *total* iff $\text{Domain}(f) = \mathbb{N}^n$ (i.e. if f is always defined when all its arguments are defined).

Definition: A *Register Machine* (abbreviated RM) is a computer model specified by a program $\mathcal{P} = \langle c_0, ..., c_{h-1} \rangle$, consisting of a finite sequence of *commands* (described below).

Intuitively, the commands operate on registers $R_1, R_2, ...$, each capable of storing an arbitrary natural number.

The possible commands are

command $R_i \leftarrow 0$	Abbreviation: Z_i	$i = 1, 2 \cdots$
$R_i \leftarrow R_i + 1$	S_i	$i = 1, 2 \cdots$
goto k if $R_i = R_j$	$J_{i,j,k}$	$i, j = 1, 2, \cdots$ and $k = 0, 1, 2, \cdots, h$

Example of a program Copy: $R_j \leftarrow R_i$
(Here i and j should be specific numbers from $\{1, 2, 3, \cdots\}$.)

c_0: $R_j \leftarrow 0$	Z_j
c_1: goto 4 if $R_i = R_j$	J_{ij4}
c_2 $R_j \leftarrow R_j + 1$	S_j
c_3 goto 1 if $R_1 = R_1$	J_{111}
c_4	

Formally, we write the above program as $\langle Z_j, J_{ij4}, S_j, J_{111} \rangle$

Semantics of RM's

A *state* is an $m+1$-tuple $\langle K, R_1, ..., R_m \rangle$ of natural numbers, where intuitively K is the instruction counter (i.e. the number of the next command to be executed) and $R_1, ..., R_m$ are the current values of the registers. (Here m must be as least as large as any register index referred to in the associated program.) Given a state $s = \langle K, R_1, ..., R_m \rangle$ and a program $\mathcal{P} = \langle c_0, ..., c_{h-1} \rangle$, the next state $s' = Next_{\mathcal{P}}(s)$ is intuitively the state resulting when command c_K is applied to the register values given by s. We say that s is a *halting* state if $K = h$, and in this case $s' = s$.

Example: Suppose the state $s = \langle K, R_1, \cdots, R_m \rangle$ and the command c_K is S_j, where $1 \leq j \leq m$. Then

$$Next_{\mathcal{P}}(s) = \langle K+1, R_1, \cdots, R_{j-1}, R_j + 1, R_{j+1}, \cdots, R_m \rangle$$

Exercise 1 *Give a formal definition of the function $Next_{\mathcal{P}}$ for the cases in which c_K is Z_i and c_K is $J_{i,j,k}$.*

A computation of a program $\mathcal{P}$ is a finite or infinite sequence $s_0, s_1, ...$ of states such that $s_{i+1} = Next_{\mathcal{P}}(s_i)$ for each s_{i+1} in the sequence. If the sequence is finite, then the last state must be a halting state, and in this case we say that the computation is *halting*. We say that a program $\mathcal{P}$ *halts* starting in state s_0 if there is a halting computation of $\mathcal{P}$ starting in state s_0.

Input/Output conventions: A program $\mathcal{P}$ computes a (partial) function $f(a_1, \ldots, a_n)$ as follows. Initially place $a_1, ..., a_n$ in $R_1, ..., R_n$ and set all other registers to 0. Start execution with command c_0. That is, the initial state is

$$s_0 = \langle 0, a_1, ..., a_n, 0, ..., 0 \rangle$$

If $\mathcal{P}$ halts starting in state s_0, the final value of R_1 must be $f(a_1, ..., a_n)$ (which then must be defined). If $\mathcal{P}$ fails to halt, then $f(a_1, ..., a_n) = \infty$.

Thus for each program $\mathcal{P}$ and each $n \geq 0$ we associate an n-ary function $f_{\mathcal{P},n}$: namely the n-ary function computed by $\mathcal{P}$.

Definition: If f is an n-ary function, we say that f is *RM-computable* (or just *computable*) if f is computed by some RM program.

Our form of *Church's Thesis:*

Every algorithmically computable function is RM-computable.

Here the notion "algorithmically computable" is not a precise mathematical notion, but rather an intuitive notion.

Church's Thesis will be discussed further at the end of this section, after we have given many examples of computable functions.

Exercise 2 *Show* $\mathcal{P} = \langle J_{234}, S_1, S_3, J_{110} \rangle$ *computes* $f(x, y) = x + y$.

Exercise 3 *Write register machine programs to compute each of the following functions:*

$f_1(x) = x \dot{-} 1$
$f_2(x, y) = x \cdot y$

Be sure to respect our input/output conventions for RM's.

Primitive Recursive Functions

Primitive recursion is a simple form of recursion defined as follows:

Definition f is defined from g and h by *primitive recursion* iff

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

We allow $n = 0$ so $\vec{x}$ could be missing.

As an example, $f_+(x, y) = x + y$ can be defined by primitive recursion as follows:

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

In this case, $g(x) = x$, and $h(x, y, z) = z + 1$.

If f is defined from g and h by primitive recursion, we can compute f from g and h by the following high-level program:

$u \leftarrow g(\vec{x})$
for $z : 0..y-1$
    $u \leftarrow h(\vec{x}, z, u)$
end for

The final value of u is $f(\vec{x}, y)$

Definition f is defined from g and $h_1 \cdots h_m$ by *composition* iff
$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_m(\vec{x}))$$
where $f, h_1 \ldots h_m$ are each n-ary functions, g is m-ary.

Initial Functions:

Z	0-ary function equal to 0	
S	$S(x) = x + 1$	
$I_{n,i}(x_1 \cdots x_n) = x_i$	$1 \leq i \leq n$	infinite class of *projection* functions

Definition f is *primitive recursive* iff f can be obtained from the initial functions by finitely many applications of primitive recursion and composition.

Examples:
$f_+(x, y)$ (see above) is primitive recursive, since it is defined from primitive recursion from g and h, where $g = I_{1,1}$ and h can be defined by composition as follows:

$$h(x, y, z) = S(I_{3,3}(x, y, z))$$

As another example, let Z_1 be the unary zero function, so $Z_1(x) = 0$ for all x. Then we can define Z_1 by primitive recursion $Z_1(0) = Z = g$, $Z_1(y+1) = Z_1(y) = h(y, Z_1(y))$; here $h = I_{2,2}$.

All constant functions $K_{n,i}(x_1, \cdots, x_n) = i$ are primitive recursive. For example, $K_{2,3}(x, y) = 3$, and we show it is primitive recursive by repeated composition as follows:

$$K_{2,3}(x, y) = S(S(S(Z_1(I_{2,1}(x, y)))))$$

Exercise 4 *Show that $x \cdot y$ and x^y are each primitive recursive functions of x and y. Show that $SQ(x) = x^2$ is primitive recursive.*

Proposition: Every primitive recursive function is total. (f is total if $f(\vec{x}) \neq \infty$ for all $\vec{x} \in \mathbb{N}^n$.)

Proof: The initial functions are all total, and the two operations composition and primitive recursion preserve totality. Hence all primitive recursive functions are total. (Formally the proof is by induction on the definition of primitive recursive function; see the following proof.)

Theorem: Every primitive recursive function is computable (on a RM).

Proof: We show that each primitive recursive function f is computable by a program which upon halting leaves all registers 0 except R_1 (which contains the output). We do this by induction on the definition of primitive recursive function. That is, the proof is by induction on the number of applications of composition and primitive recursion needed to derive f from the initial functions.

The base case: Each initial function is easily shown to be computable by such a program.

Exercise 5 *For each initial function, give an RM program which computes it and leaves all registers 0 except R_1.*

Induction Step: We show that composition and primitive recursion each preserve computability by such programs.

a) Composition: Assume that $g, h_1, ..., h_m$ are computable by programs $\mathcal{P}_g, \mathcal{P}_1, ..., \mathcal{P}_m$, respectively, where these programs leave all registers 0 except R_1. We are to show that f is computable by such a program $\mathcal{P}_f$, where

$$f(\vec{x}) = g(h_1(\vec{x}), \cdots, h_m(\vec{x}))$$

At the start $x_1, \cdots, x_n$ are in registers $R_1, \cdots, R_n$, with all other registers 0. Program $\mathcal{P}_f$ proceeds as follows:

Copy $x_1, \cdots, x_n$ to $R_{1+k}, \cdots, R_{n+k}$, where k is larger than n and larger than any register index referred to in any of the programs $\mathcal{P}_1, ..., \mathcal{P}_m$. Now compute $h_1(\vec{x}), \cdots, h_m(\vec{x})$ and store them in registers $R_{n+k+1}, ..., R_{n+k+m}$. Do this by executing copies of each program $\mathcal{P}_1, ..., \mathcal{P}_m$ in turn, where each jump instruction $J_{i,j,r}$ in each program is replaced by $J_{i,j,r+d}$, where d is the number of commands in $\mathcal{P}_f$ preceding the first command of the copy. (Note that by the definitions on page 53 the only way an RM program can halt is to jump to h, which is one beyond the last command.) After computing each $h_i(\vec{x})$ copy the result in R_1 to R_{n+k+i} and restore $R_1, ..., R_n$ to $x_1, ..., x_n$ by copying $R_{1+k}, ..., R_{n+k}$ back to $R_1, ..., R_n$.

After computing $h_1(\vec{x}), ...h_m(\vec{x})$, copy $R_{n+k+1}, ..., R_{n+k+m}$ to $R_1, ..., R_m$, and then set all other registers to 0. Finally execute a copy of $\mathcal{P}_g$, with jump instructions suitably modified as above.

b) Primitive Recursion: Implement the informal algorithm given after the definition of primitive recursion.

Exercise 6 *Give an RM program to implement primitive recursion.*

Is the converse true? Is every computable function primitive recursive?
No. Some computable functions are not total.

Is every *total* computable function primitive recursive?

No, we can give a "diagonal" argument to show this. Here we give an outline of this argument.

The first step is to assign a natural number e to every description of a primitive recursive function. The description specifies exactly how the function is obtained from the initial functions by the operations primitive recursion and composition. This description is a string of symbols, and the number is assigned to the string in some standard fashion (this will be discussed later.) Let f_e be the function whose description is coded by e. We are interested here in the unary functions (functions of one variable), so define g_e to be f_e if f_e is a unary function, and let g_e be the constant zero function otherwise. Thus

$$g_0, g_1, g_2, \cdots$$

is an effective enumeration of the set of unary primitive recursive functions. Here the word "effective" means that if we define the "universal" function U by

$$U(x, y) = f_x(y)$$

then the binary function U is computable.

We will show that U is not primitive recursive. For suppose that U is primitive recursive. Then the "diagonal" function $D(x)$ defined by

$$D(x) = g_x(x) + 1$$

is primitive recursive. But then $D = g_e$ for some e, so

$$g_e(e) = D(e) = g_e(e) + 1$$

which is a contradiction. □

Now we give another example.

Ackermann's Function: A total computable function not primitive recursive

Let $A_0(x) = \begin{cases} x+1 & \text{if } x = 0 \text{ or } x = 1 \\ x+2 & \text{otherwise} \end{cases}$

$$\begin{cases} A_{n+1}(0) = 1 \\ A_{n+1}(x+1) = A_n(A_{n+1}(x)) \end{cases}$$

Now let $A(n, x) = A_n(x)$. We can prove by induction on n that A_n is a total function. Therefore A is a total function. Also A is intuitively computable because the equations above represent a recursive program for computing A. It follows from Church's Thesis (page 54) that A is RM-computable. (It is an interesting exercise to write an RM program to compute A.) We argue below that A is not primitive recursive.

$A_1(x) = 2x$ for $x > 0$, since $A_1(0) = 1$ and $A_1(1) = 2$ (check this) and for $x > 0$ $A_1(x+1) = A_0(A_1(x))$, so $A_1(x+1) = A_1(x) + 2$ for $x > 0$.

$A_2(x) = A_1(A_1(...(A_1(1))) = 2^x$

$A_3(x) = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\} x$ 2's

$A_4(4)$ is so large it would fill the universe with digits.

Lemma: For each n, A_n is primitive recursive.

Proof: Induction on n

The base case is to show A_0 is primitive recursive. This will be easy after we show that the primitive recursive functions are closed under definition by cases later on.

The induction step is easy. $\square$

Fact For every primitive recursive function $h(\vec{x})$ there exists n so that $A_n(x)$ *dominates* $h(\vec{x})$. Here $A_n(x)$ *dominates* $h(\vec{x})$ means that there is a constant B such that for all $x_1, \cdots, x_k \geq B, \quad A_n(\max\{x_1, \cdots, x_k\}) > h(x_1, \cdots, x_k)$. (This can be proved by induction on the definition of primitive recursive function.) Thus if we define $A(n, x) = A_n(x)$, then $A(n, x)$ is *not* a primitive recursive function of both n and x. In fact $F(x) = A(x, x)$ is not primitive recursive, since A cannot dominate itself. Explicitly, there is no fixed pair of numbers n, B such that $A(n, x) > F(x)$ for all $x \geq B$. For otherwise we could set $m = \max(n, B)$, and then $A(n, m) > F(m) = A(m, m) \geq A(n, m)$, a contradiction. (Here we use the fact that $A(x, y)$ is monotone nondecreasing in both x and y.)

We are trying to characterize the computable functions. Composition and primitive recursion are not enough, so we need another operation.

Minimization

We let μ denote the least number operator. More precisely:

Definition: $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$
iff 1) $f(\vec{x})$ is the least number b such that $g(\vec{x}, b) = 0$
2) $g(\vec{x}, i) \neq \infty$ for $i < b$
$f(\vec{x}) = \infty$ if no such b exists

Notice that $f(\vec{x})$ may be undefined for some values of $\vec{x}$, even though g is a total function.

Lemma: If g is computable and $f(\vec{x}) = \mu y[g(\vec{x}, y) = 0]$ then f is computable.

Proof: Given an RM program for g, we need to construct a program for f.

High level version: for $y = 0..\infty$
if $g(\vec{x}, y) = 0$
then output y, exit
end if

end for

□

Exercise 7 *Give the RM program which computes f.*

Notice that if $g(\vec{x}, y) = \infty$ for some y smaller than the least b such that $g(\vec{x}, b) = 0$, then the above program does not terminate. That is why we need clause 2) in the definition of μ.

Recursive Functions

Definition: f is *recursive* iff f can be obtained from the initial functions by finitely many applications of composition, primitive recursion, and minimization.

It follows immediately from the above definition that every primitive recursive function is recursive.

Theorem: Every recursive function is computable.

Proof: Induction on the definition of recursive function. (I.e., induction on the number of applications of the operators composition, primitive recursion, and minimization needed to derive the function.) We have already done all the steps. □

It turns out that the converse is also true: All computable functions are recursive (Kleene 1940's). This requires more work to prove. We will start by showing lots of functions are primitive recursive. Anticipating a little, it turns out that any function computable in exponential time is primitive recursive. In fact if f can be computed in time $T(x) = O(A_m(x))$ for some fixed m (this is Ackermann's function), then f is primitive recursive. For example, if the time of the algorithm computing f is bounded above by

$$\left.2^{2^{2^{\cdot^{\cdot^{2}}}}}\right\}n$$

then f is primitive recursive.

We now show that lots of functions are primitive recursive. We know $+$, $\cdot$, 2^x, x^y are primitive recursive (see exercise (4)).

Next we show that the predecessor function pd is primitive recursive where
$pd(x) = \begin{cases} x-1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$

We can define pd by primitive recursion as follows:

$pd(0) = 0 = g$
$pd(x+1) = x = h(x, pd(x))$, where

$h(x,z) = x, \quad h = I_{2,1}$ (projection), $g = Z$ (initial function)

Limited subtraction: Define

$$x \mathbin{\dot{-}} y = \begin{cases} x - y & \text{if } y \le x \\ 0 & \text{if } y > x \end{cases}$$

We can define this by primitive recursion from pd as follows:

$$x \mathbin{\dot{-}} 0 = x$$
$$x \mathbin{\dot{-}} (y+1) = pd(x \mathbin{\dot{-}} y)$$

Now we can show $\max(x,y)$ is primitive recursive by taking the composition of two primitive recursive functions as follows:

$$\max(x,y) = (x \mathbin{\dot{-}} y) + y$$

Relations considered as total 0-1 valued functions

Let $R \subseteq \mathbb{N}^n$. Thus R is an n-ary relation (predicate). We will think of R as a total 0-1 valued function as follows: $R(\vec{x}) = 0$ iff $\vec{x} \in R$, and $R(\vec{x}) = 1$ otherwise.

Thus a relation R is *primitive recursive* (respectively, *computable*) iff the corresponding total 0-1 valued function is primitive recursive (respectively, computable).

Note that in this course, 0 = "true" and 1 = "false", in contrast to computer science conventions. This convention of using 0 to stand for "true" among logicians goes back at least to Gödel.

Lemma: if R and S are n-ary primitive recursive (resp. computable) predicates, then so are $\neg R$, $R \vee S$, and $R \wedge S$.

Proof: For this we will define another useful primitive recursive function:

Definition: $\overline{sg}(x) = 1 \mathbin{\dot{-}} x$

Thus $\overline{sg}(x) = 0$ if $x > 0$, and 1 if $x = 0$.

To prove the Lemma, note that $(\neg R)(\vec{x}) = \overline{sg}(R(\vec{x}))$
$(R \vee S)(\vec{x}) = R(\vec{x}) \cdot S(\vec{x})$
$(R \wedge S)(\vec{x}) = \neg(\neg R \vee \neg S)(\vec{x})$ $\square$

Now we can show that the following relations are primitive recursive. Define

$$(x < y) = \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \ge y \end{cases}$$

$(x < y) = \overline{sg}(y \mathbin{\dot{-}} x)$, so this is a primitive recursive relation.

Similarly

$(x = y) = \neg(x < y) \wedge \neg(y < x)$

$(x \leq y) = (x < y) \vee (x = y)$

Further operations which preserve primitive recursive functions and relations (These also preserve computable relations and computable functions)

1) Bounded sum and bounded products

   $g(\vec{x}, y) = \sum_{z<y} f(\vec{x}, z) = f(\vec{x}, 0) + \cdots + f(\vec{x}, y-1)$

   g can be defined from f by primitive recursion as follows:

   $g(\vec{x}, 0) = 0$
   $g(\vec{x}, y+1) = g(\vec{x}, y) + f(\vec{x}, y)$
   $h(\vec{x}, y) = \prod_{z<y} f(\vec{x}, z) = f(\vec{x}, 0) \cdot \ldots \cdot f(\vec{x}, y-1)$

   h can be defined from f by primitive recursion as follows:

   $h(\vec{x}, 0) = 1$
   $h(\vec{x}, y+1) = f(\vec{x}, y) \cdot h(\vec{x}, y)$

2) Bounded quantification

   $S(\vec{x}, y) = \exists z < y R(\vec{x}, z)$ (This means $\exists z(z < y \wedge R(\vec{x}, z))$)
   $T(\vec{x}, y) = \forall z < y R(\vec{x}, z)$ (This means $\forall z(z < y \supset R(\vec{x}, z))$)

   If R is a primitive recursive (resp. computable) relation, then so are S and T, because

   $S(\vec{x}, y) = \prod_{z<y} R(\vec{x}, z)$, and

   $T(\vec{x}, y) = \neg\exists z < y \neg R(\vec{x}, z)$.

As an application, we show that the divisibility relation $x|y$ (y is divisible by x) is primitive recursive:

$x|y = \exists z \leq y(x \cdot z = y)$.
(Note that $\exists z \leq y R = \exists z < (y+1) R$)

Now define

$$\begin{aligned} \text{Prime}(x) &= \begin{cases} 0 & \text{if } x \text{ is prime} \\ 1 & \text{otherwise} \end{cases} \\ &= 1 < x \wedge \forall z < x(\neg(z|x) \vee z = 1) \end{aligned}$$

Thus Prime(x) is a primitive recursive relation.

$$\begin{aligned} Cond(x, y, z) &= \begin{cases} y \text{ if } x = 0 \\ z \text{ if } x > 0 \end{cases} \\ &= \overline{sg}(x) \cdot y + sg(x) \cdot z \end{aligned}$$

Here $sg(x) = \overline{sg}(\overline{sg}(x))$. Thus $Cond(x, y, z)$ is primitive recursive.

Definition by cases

$$f(\vec{x}) = \begin{cases} g(\vec{x}) \text{ if } R(\vec{x}) \\ h(\vec{x}) \text{ otherwise} \end{cases}$$

Then $f(\vec{x}) = Cond(R(\vec{x}), g(\vec{x}), h(\vec{x}))$, so f is primitive recursive (resp. computable) if g, h, and R are primitive recursive (resp. computable).

Exercise 8 *For each unary relation $R(x)$ define the unary function $\#R(x)$ by*

$$\#R(x) = |\{y \leq x : R(y)\}|$$

where $|S|$ is the number of elements in a set S.

(a) Show that if R is primitive recursive, then $\#R$ is primitive recursive.

(b) Let $\pi(x)$ be the number of prime numbers $\leq x$. For example $\pi(6) = 3$, since $\{2, 3, 5\}$ comprise the set of primes ≤ 6. Prove that $\pi(x)$ is primitive recursive.

Bounded Minimization

Let $R(\vec{x}, y)$ be a relation (i.e. a total, 0-1 valued function.) We use the notation

$$f(\vec{x}, y) = \min z < yR(\vec{x}, z) = \begin{cases} \text{least } z \text{ so } z < y \text{ and } R(\vec{x}, z) \text{ if such } z \text{ exists} \\ y \text{ otherwise} \end{cases}$$

Notice that f is always a total function. This is because by convention every relation $R(\vec{x}, z)$ is a total 0-1 valued function, and we have defined a default value y in case there is no $z < y$ satisfying $R(\vec{x}, z)$. This is in contrast to the result of applying the unbounded minimization operator μ. That is, $\mu z R(\vec{x}, z)$ could be nontotal, even though R is always total.

Suppose that $f(\vec{x}, y) = \min z < yR(\vec{x}, z)$, as above. In order to show that f is a primitive recursive (resp. total computable) function whenever R is a primitive recursive (resp. computable) relation, it is sufficient to establish the clever identity

$$f(\vec{x}, y) = \sum_{z<y} (\exists v \leq z R(\vec{x}, v))$$

To see why this holds, study the table below:

$$\text{say } m = \min z < yR(\vec{x}, z) \qquad \begin{array}{c|l} z & (\exists v \le zR(\vec{x}, v)) \\ \begin{array}{c} 0 \\ 1 \\ \vdots \\ m-1 \end{array} & \left.\begin{array}{c} 1 \\ 1 \\ \vdots \\ 1 \end{array}\right\} m \text{ 1's } \therefore \text{ sum is } m \\ m & \ 0 \\ m+1 & \ 0 \\ \vdots & \ \vdots \end{array}$$

Exercise 9 *Give a suitable definition of* $\max z < yR(\vec{x}, z)$ *and show that if*

$$f(\vec{x}, y) = \max z < yR(\vec{x}, z)$$

then f *is primitive recursive if* R *is primitive recursive.*

Applications of the above operations:

We can show that the integer quotient function $q(x, y) = \lfloor x/y \rfloor = x \text{ div } y$ and the remainder function $rm(x, y) = x \bmod y$ are primitive recursive:

$q(x, y) = \min z \le x[y(z+1) > x]$
$rm(x, y) = x \dot{-} (y \cdot q(x, y))$

Note that the first equation gives $q(x, 0) = x + 1$. This convention for the case $y = 0$ does not matter, since in practice we will never want to divide by 0.

Note also that these definitions represent inefficient, exponential time algorithms, but we are not concerned about complexity here.

Now we show that $f(i) =$ the ith prime is a primitive recursive function of i, because it can be defined by primitive recursion as follows:

$p_0 = 2$
$p_{x+1} = \min y < p_x^{p_x} \ (p_x < y \wedge \text{Prime}(y))$

Thus $p_0, p_1, p_2, \cdots$ is an enumeration of all prime numbers, in increasing order.

Note that the upper bound $p_x^{p_x}$ on p_{x+1} given above exceeds Euclid's upper bound of $p_x!$.

Prime Decomposition of Numbers

Recaull that the Unique Factorization Theorem for $\mathbb{N}$ states that for every $x > 0$ there is a unique sequence $e_0, e_1, \ldots$ of natural numbers such that

$$x = p_0^{e_0} p_1^{e_1} p_2^{e_2} \cdots$$

where only finitely many of the exponents e_i are nonzero.

For example,

$$63 = 2^0 \cdot 3^2 \cdot 5^0 \cdot 7^1 \cdot 11^0 \cdot \dots = p_0^0 \cdot p_1^2 \cdot p_2^0 \cdot p_3^1 \cdot p_4^0 \cdot \dots$$

Notation: $(z)_x$ = exponent of p_x in prime decomposition of z. Thus

$$z = p_0^{(z)_0} p_1^{(z)_1} \cdots p_m^{(z)_m}$$

$(z)_x = \min y < z(\neg p_x^{y+1} \mid z)$

so $(z)_x$ is a primitive recursive function of z and x.

length: $lh(z) = \min y < z(\prod_{x<y} p_x^{(z)_x} = z)$ is prim rec

$= 1 +$ subscript of largest prime that divides z

Simulating RM's

Now we have enough general machinery to show many functions are primitive recursive. We will use primitive recursive functions to simulate a RM.

First we assign a "Gödel number" $\#(\mathcal{P})$ to program $\mathcal{P}$.

commands	c	Z_i	S_i	J_{ijm}
codes	$\#(c)$	2^i	3^i	$5^i 7^j 11^m$

because of prime decomposition of numbers these numbers are unique

If the program $\mathcal{P} = \langle c_0, \dots, c_{h-1} \rangle$, then

$$\#(\mathcal{P}) = p_0^{\#(c_0)} p_1^{\#(c_1)} \cdots p_{h-1}^{\#(c_{h-1})}$$

Note that distinct programs get distinct codes.

This idea of using prime powers to assign unique numerical codes to combinatorial objects such as programs goes back to Gödel in 1930. From a computer science point of view, the numbers seem ridiculously large, and in fact more efficient methods would use ASCII style codes, or pairing functions. However our present purpose is simply to show that various functions are primitive recursive, so we will not worry about practical computability.

Define the relation $\text{Succ}(x, i) = (x$ codes the command $R_i \leftarrow R_i + 1)$ (i.e. the command S_i). This is the relation $(x = 3^i)$, and therefore it is primitive recursive. Similarly:

$\text{zero}(x, i) = (x = 2^i)$ $\quad x$ codes $R_i \leftarrow 0$
$\text{Jump}(x, i, j, k) = (x = 5^i 7^j 11^k)$ $\quad x$ codes J_{ijk}

$\text{Prog}(z) = (z = \#(P))$ for some program P
$\text{Prog}(z) = (\forall j < lh(z)[\text{Command}((z)_j)])$, where
$\text{Command}(x) = \exists i < x(\text{Succ}(x, i) \vee \text{zero}(x, i) \vee \exists j < x \exists k < x(\text{Jump}(x, i, j, k)))$

Let $\mathcal{P}$ be a RM program which mentions at most registers $R_1, \cdots, R_m$, that is an upper bound on the number of registers is m

Recall that a *state* of $\mathcal{P}$ is a tuple $(K, R_1 \cdots R_m)$, where
- K = line # of next command to be executed (instruction counter)
- R_i = current contents of ith register

We code this state by $p_0^K p_1^{R_1} \cdots p_m^{R_m} = u$

More definitions:

$$\hat{z} = \begin{cases} z \text{ if } Prog(z) & \text{i.e. } z \text{ is the number of a program} \\ 1 \text{ otherwise} \end{cases}$$

Notation: $\{z\}$ = the program P s.t. $\#(P) = \hat{z}$

Thus $\{z\} = \begin{cases} \text{the program } P \text{ such that } \#(P) = z \text{ if } P \text{ exists} \\ \Lambda \text{ (empty program) otherwise} \end{cases}$

Definition: The function Nex is defined by the condition $Nex(u, z) = u'$ iff u' codes the state resulting when program $\{z\}$ executes one step when the current state is coded by u. If u codes a halting state, then $u' = u$.

Lemma: Nex is primitive recursive.

Proof Outline: Nex can be defined by a large definition by cases, where the cases depend on the command $(z)_K$, where $K = (u)_0$. We leave the details as a long exercise. □

Exercise 10 *Carry out the details in the above proof.*

Computations

Recall the definition of computation given at the beginning of this section under Semantics of RM's. If $(u_0, u_1, ..., u_t)$ is the sequence of codes for the successive states in a computation, then we code the entire computation by the number

$$y = p_0^{u_0} p_1^{u_1} \cdots p_t^{u_t}$$

$halt(u, z)$ holds iff u codes a halting state of program $\{z\}$. Then $halt(u, z)$ holds iff

$$(u)_0 \geq lh(\hat{z})$$

and hence *halt* is a primitive recursive relation. (Recall $(u)_0 = K$ is the instruction counter.)

Kleene T predicate (Important)

Definition: For each $n \geq 1$ we define the n+2-ary relation T_n by the condition $T_n(z, x_1, \cdots, x_n, y)$ holds iff y codes the computation of program $\{z\}$ on input $\vec{x}$. For $n = 1$ we sometimes write T instead of T_1.

Theorem: (Kleene) For each $n \geq 1$ T_n is a primitive recursive relation.

Proof: $T_n(z, \vec{x}, y)$ holds iff y codes a computation $(u_0, ..., u_t)$, where the initial state u_0 satisfies

$$u_0 = p_0^0 p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n}$$

and for all $i < t$

$$u_{i+1} = Nex(u_i, z)$$

and $halt(u_t, z)$ (the last state is halting) and for all $i < t$

$$\neg halt(u_i, z)$$

(no intermediate state is halting).

More formally, setting $t = lh(y) \dot{-} 1$,

$$\begin{aligned} T_n(z, \vec{x}, y) = [(y)_0 = p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n} & \quad \{\text{initial state}\} \\ \wedge \forall i < t[(y)_{i+1} = Nex((y)_i, z)] & \\ \wedge halt((y)_t, z) \ \{\text{last state is halting}\} & \\ \wedge \forall i < t \neg halt((y)_i, z) \ \{\text{no intermediate state is halting}\} \ \Box & \end{aligned}$$

Output function

We define the output function $U(y)$ to be the contents of register R_1 in the final state of the computation coded by program $\{y\}$.

Thus $U(y) = ((y)_{lh(y) \dot{-} 1})_1$, and hence U is primitive recursive.

Notation: $\{z\}_n$ is the n-ary function computed by program $\{z\}$

Kleene Normal Form Theorem: There is a primitive recursive function U and for each $n \geq 1$ a primitive recursive predicate T_n such that

$$\{z\}_n(x_1, \ldots, x_n) = U(\mu y T_n(z, \vec{x}, y))$$

Proof: Immediate from the definitions above. Note: The least y is the only y satisfying the condition $T_n(...)$. Also note that y will not exist if the program doesn't halt, so $\{z\}_n(\vec{x})$ is undefined in this case.

Corollary: Every computable function is recursive, and can be obtained using at most one application of μ.

Universal Functions

Notation: $\Phi_n(z, \vec{x}) = \{z\}_n(\vec{x})$

Φ_n is called a universal function, since it codes every computable function of n variables, as z varies.

Corollary: The universal function Φ_n is recursive (and hence computable), for $n = 1, 2, ...$

A program computing Φ_n is called an *interpreter.*

The function Φ_1 is universal for the set of all unary computable functions. Thus if we define $\phi_i(x) = \Phi_1(i,x)$ then

$$\phi_0, \phi_1, ...$$

is an enumeration of all (partial) unary computable functions. It turns out that it is essential to include nontotal functions in order to get a computable universal function.

Theorem: There is no computable universal function for the set of all *total* computable unary functions.

Proof: Let $f_0, f_1, ...,$ be a list of all total computable unary functions, in any order, possibly with repetitions. Let

$$F(z,x) = f_z(x)$$

We will show that F is not computable. This is because if F were computable, then the "diagonal" function

$$D(x) = F(x,x) + 1 = f_x(x) + 1$$

would also be a total computable unary function. But then D must be in the list $f_0, f_1, ...$. That is, $D = f_e$ for some e. But then $f_e(e) = D(e) = f_e(e) + 1$, a contradiction. Hence F is not computable.

Exercise 11 *Prove that there is no computable universal relation $RU(x,y)$ for all computable unary relations.*

Exercise 12 *Let $A(n,x) = A_n(x)$ be Ackermann's Function (page 57). Define*

$$UP(z,x) = U(\min y < (A((z)_0, x) + z)\ T((z)_1, x, y))$$

where $T(z,x,y)$ is the Kleene T-predicate. (See page 64 for the notation $(z)_x$.) (Compare the definition of UP with the Kleene Normal Formal Theorem. Use the facts stated about Ackermann's function together with results above to prove the following.

(a) Prove that UP is a total computable function.

(b) Prove that for each $e \in \mathbb{N}$, the unary function $g_e(x) = UP(e,x)$ is primitive recursive.

(c) Prove that for each unary primitive recursive function $f(x)$ there is $e \in \mathbb{N}$ such that $f = g_e$ (where g_e is defined in part (b)). Use the Kleene Normal Form Theorem, and the following strengthening of the Theorem, page 55:

Fact: *Every primitive recursive function $f(\vec{x})$ is computable by a RM program $\mathcal{P}$ such that the function $Comp_{\mathcal{P}}(\vec{x})$ is primitive recursive, where*

$$Comp_{\mathcal{P}}(\vec{x}) \text{ is the number coding the computation of } \mathcal{P} \text{ on input } \vec{x}$$

(d) Give a diagonal argument showing that UP is not primitive recursive (see page 57 for an outline).

Exercise 13 *Use a diagonal argument to prove that*

$$H(x) = \mu y T(x, x, y)$$

has no total computable extension. That is, show that if the function $BIG(x)$ is total and agrees with $H(x)$ whenever $H(x)$ is defined, then $BIG(x)$ is not computable.

Church's Thesis: Every "algorithmically computable function" is computable (i.e. computable on a RM).

This statement cannot be proved because it is not precise. But it is a strong claim about the robustness of our formal notion of computable function. In general, if we give an informal algorithm to compute a function, then we can claim that it is computable, by Church's Thesis.

Alonzo Church proclaimed this famous "thesis" in a footnote to a paper in 1936. Actually, he did not talk about RM's, but rather claimed that every algorithmically computable function is definable using the λ-calculus which he had invented. A little later Alan Turing published his famous paper defining what are now called Turing machines, and argued, more convincingly than Church, that every algorithmically computable function is computable on a Turing machine. (Hence "Church's Thesis" is sometimes called the "Church-Turing thesis".) Turing proved that Church's λ-definable functions coincide with the Turing computable functions. It turns out that both of these coincide with the functions computable on a RM, which we have shown coincide with the recursive functions. In fact, many other formalisms for defining algorithmically computable functions have been given, and all of them turn out to be equivalent. This robustness is a powerful argument in favour of Church's thesis.