

Chapter 2

Recursive Functions

In this chapter, we consider the notion of a computable function $f : \mathbb{N}^n \rightarrow \mathbb{N}$. Such a function is computable if there is a finite set of instructions for a procedure which, if followed on input (x_1, \dots, x_n) , terminates with output $f(x_1, \dots, x_n)$ (for example, a computer program). No restriction is made on the time or space required in the device used to implement the procedure. (This, of course, is unrealistic, but it is easier to develop a theory without such restrictions.)

More generally, we consider *partial functions* $f : \mathbb{N}^n \rightarrow \mathbb{N}$. Recall that this means f is a function $X \rightarrow \mathbb{N}$ where X is a subset of \mathbb{N}^n . Such a function is computable if such a set of instructions exists, but the procedure terminates with output $f(x_1, \dots, x_n)$ if this is defined, and otherwise does not terminate. (Also recall that, in this context, a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$, defined on all of \mathbb{N}^n , is called a *total function*.)

This idea of computability cannot be subjected to a mathematical analysis because various terms, such as “procedure” have not been precisely defined. Nevertheless, it is possible to make some progress even with such a vague notion. As an example, take the following statement. Suppose $g, h : \mathbb{N} \rightarrow \mathbb{N}$ are two computable functions; then their composition $g \circ h$ is computable (recall that $(g \circ h)(n) = g(h(n))$). To “prove” this, take a procedure which computes h , give it input n , then pass the output to a procedure which computes g . The output is $g(h(n))$, so this is a procedure to compute $g \circ h$.

To obtain a mathematical theory, the way to proceed is to develop this idea. Write down a collection of functions which one expects to be computable (under any reasonable definition). Then give ways of constructing new functions which, applied to computable functions, should lead to new computable functions (such as composition, as described above). Then take all functions obtained from the initial functions by repeated use of these operations. This is what we shall do, leading to a class of functions called *partial recursive functions*.

We have to hope that we have written down enough initial functions and ways of constructing new functions that all possible computable functions are recursive. This is, of course, impossible to prove. Nevertheless, the assertion that this is true has a name.

Church's Thesis. The partial computable functions as described above are precisely the partial recursive functions.

This is sometimes called the Church-Turing thesis. In practice, it is used like the word “clearly” in other branches of mathematics. Thus “ f is partial recursive by Church's thesis” means “ f is obviously computable and I don't want to write out the lengthy details needed to prove it's recursive”. We shall not use it in this way.

As evidence for Church's thesis, we shall consider several precise notions of computability and show that, in all cases, the partial computable functions are precisely the partial recursive functions. We shall consider computability by register programs. These resemble programs in a very simple assembly language. The machine which executes these programs has “registers”, each of which can store any natural number, which can be changed when the program runs. This unrealistic assumption is compounded by making no limit on the number of registers a program may use, so the machine is given infinitely many registers. This reflects the statement made above in introducing computable functions: no restriction is made on the time or space required. Thus the machine implementing the program is expected to continue indefinitely without running out of power or breaking down.

We also consider computability by *abacus machines*, which can be viewed as versions of register programs written in a higher-level language, where only well-structured programs are possible. Finally, we discuss computability by Turing machines, a new use for them after their use in language recognition in Chap. 1.

Before defining the class of partial recursive functions, it is useful to define a smaller class, the class of *primitive recursive functions*, which are all total. Many standard functions on the natural numbers are primitive recursive. The definition involves just two ways of constructing new functions; a generalisation of composition discussed above, and “primitive recursion”. We shall define these for arbitrary partial functions, so that no modification is needed when defining the partial recursive functions. Also, it is convenient to introduce the idea of a primitively recursively closed class, so that our results apply to other classes, such as the class of recursive functions defined later on.

Definition. Let $g : \mathbb{N}^r \rightarrow \mathbb{N}$, $h_1, \dots, h_r : \mathbb{N}^n \rightarrow \mathbb{N}$ be partial functions. The function $f = g \circ (h_1, \dots, h_r)$ obtained from g, h_1, \dots, h_r by *composition* is the partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ defined by

$$f(x_1, \dots, x_n) = g(h_1(x_1, \dots, x_n), \dots, h_r(x_1, \dots, x_n))$$

where the left-hand side of the equation is defined if and only if the right-hand side is.

If g, h_1, \dots, h_r are computable functions, one can see that f is computable by a simple generalisation of the discussion above.

Definition. Let $g : \mathbb{N}^n \rightarrow \mathbb{N}$, $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ be partial functions. The function $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ obtained from g and h by *primitive recursion* is defined by:

$$f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n)$$

$$f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)).$$

For a formal proof that these equations do define a unique partial function f , we refer to [4, §3.7]. For given (x_1, \dots, x_n) , $f(x_1, \dots, x_n, y)$ is defined either for no y , for all y , or for $0 \leq y \leq r$ for some r . Note that $n = 0$ is allowed, when g is viewed as a fixed natural number.

If g and h are computable, then so is f . Given $\underline{x} = (x_1, \dots, x_n)$, we first use a procedure to compute $g(\underline{x})$. If it terminates, the value obtained is $f(\underline{x}, 0)$. We can then use this value and a procedure to compute h to find $f(\underline{x}, 1)$. If this terminates, we can then use the computed value of $f(\underline{x}, 1)$ and the procedure to compute h to compute $f(\underline{x}, 2)$, and so on.

We also define the *initial functions* to be the functions in the following list:

- (zero function) $z : \mathbb{N} \rightarrow \mathbb{N}$ defined by $z(x) = 0$ for all $x \in \mathbb{N}$
- (successor function) $\sigma : \mathbb{N} \rightarrow \mathbb{N}$ defined by $\sigma(x) = x + 1$
- the *projection functions* $\pi_{in} : \mathbb{N}^n \rightarrow \mathbb{N}$ defined by $\pi_{in}(x_1, \dots, x_n) = x_i$ (for $n \geq 1$ and $1 \leq i \leq n$).

The initial functions are all computable; it is left to the reader to justify this. We now define

$$\mathcal{P} = \{f \mid \text{for some } n > 0, f \text{ is a partial function } \mathbb{N}^n \rightarrow \mathbb{N}\}$$

$$\text{and } \mathcal{T} = \{f \in \mathcal{P} \mid f \text{ is total}\}$$

In this chapter, a *class of functions* means a subset of \mathcal{P} and a *class of total functions* means a subset of \mathcal{T} .

Definition. A class of total functions \mathcal{C} is *primitively recursively closed* if

- (1) \mathcal{C} contains all the initial functions;
- (2) \mathcal{C} is closed under composition (i.e. if f is obtained from g, h_1, \dots, h_r by composition, and g, h_1, \dots, h_r are all in \mathcal{C} , then $f \in \mathcal{C}$);
- (3) \mathcal{C} is closed under primitive recursion (i.e. if f is obtained from g and h by primitive recursion, and $g, h \in \mathcal{C}$, then $f \in \mathcal{C}$).

There is a smallest primitively recursively closed class (the intersection of all primitively recursively closed total classes), called the class of *primitive recursive functions*.

Note. It is left to the reader to show that a function f is primitive recursive if and only if there is a sequence $f_0, \dots, f_k = f$ of functions, where each f_i is either an initial function, or is obtained by composition from some of the f_j , for $j < i$, or is obtained by primitive recursion from two of the f_j with $j < i$. Such a sequence is called a *primitive recursive definition* of f .

Examples of Primitive Recursive Functions.

- (1) (addition) The function $s : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined by $s(x, y) = x + y$ is primitive recursive. For

$$\begin{aligned} s(x, 0) &= g(x) \text{ where } g = \pi_{11} \text{ (the identity mapping on } \mathbb{N}) \\ s(x, y + 1) &= s(x, y) + 1 = h(x, y, s(x, y)), \text{ where } h = \sigma \circ \pi_{33} \end{aligned}$$

so $\pi_{11}, \pi_{33}, \sigma, \sigma \circ \pi_{33}, s$ is a primitive recursive definition.

- (2) (multiplication) $m : \mathbb{N}^2 \rightarrow \mathbb{N}$ defined by $m(x, y) = xy$ is primitive recursive. For

$$\begin{aligned} m(x, 0) &= 0 = z(x) \\ m(x, y + 1) &= m(x, y) + x = s(\pi_{33}(x, y, m(x, y)), \pi_{13}(x, y, m(x, y))) \\ &= h(x, y, m(x, y)), \text{ where } h = s \circ (\pi_{33}, \pi_{13}). \end{aligned}$$

From this, it is easy to write down a primitive recursive definition. (In this and subsequent examples, this will be left to the reader.)

- (3) (exponential function) $\exp(x, y) = x^y$ is primitive recursive. For

$$\begin{aligned} \exp(x, 0) &= 1 \\ \exp(x, y + 1) &= m(x, \exp(x, y)). \end{aligned}$$

- (4) (factorial) $\text{Fac}(x) = x!$ is primitive recursive since $\text{Fac}(0) = 1, \text{Fac}(x + 1) = m(x + 1, \text{Fac}(x))$.
- (5) Any constant function $\mathbb{N}^n \rightarrow \mathbb{N}$ is primitive recursive. For $n = 1$, the constant function 0 is z , the constant function 1 is $\sigma \circ z$, the constant function 2 is $\sigma \circ (\sigma \circ z)$, etc. For general n , the constant function c is $c' \circ \pi_{1n}$, where $c' : \mathbb{N} \rightarrow \mathbb{N}$ is the constant function with value c .
- (6) (predecessor) We define $\text{Pred}(x)$ to be $x - 1$ if $x > 0$ and $\text{Pred}(0)$ to be 0. This is primitive recursive since $\text{Pred}(0) = 0, \text{Pred}(x + 1) = x$.
- (7) (proper subtraction) $x \dot{-} y = \max\{x - y, 0\}$ is primitive recursive: $x \dot{-} 0 = x, x \dot{-} (y + 1) = \text{Pred}(x \dot{-} y)$.
- (8) (modulus) $|x - y| = (x \dot{-} y) + (y \dot{-} x)$ is primitive recursive.
- (9) (sign) $\text{sg}(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$ is primitive recursive, because $\text{sg}(0) = 0$ and $\text{sg}(x + 1) = 1$.

Remark 2.1. If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is in \mathcal{C} (a primitively recursively closed class) and $g : \mathbb{N}^m \rightarrow \mathbb{N}$ is defined by $g(x_1, \dots, x_m) = f(y_1, \dots, y_n)$, where each y_i is either a constant or x_j for some fixed j , then $g \in \mathcal{C}$. (For $g = f \circ (h_1, \dots, h_n)$, where h_i is either a constant function or some π_{jm} .)

Lemma 2.1. Let \mathcal{C} be a primitively recursively closed class, and let $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ be in \mathcal{C} . Then the following functions are in \mathcal{C} .

- (1) $f_1 : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, where $f_1(x_1, \dots, x_n, y) = \sum_{t=0}^y g(x_1, \dots, x_n, t)$.

$$(2) f_2 : \mathbb{N}^{n+1} \rightarrow \mathbb{N}, \text{ where } f_2(x_1, \dots, x_n, y) = \prod_{t=0}^y g(x_1, \dots, x_n, t).$$

Proof. Both f_1 and f_2 are obtained by primitive recursion from functions in \mathcal{C} , since

$$(1) f_1(\underline{x}, 0) = g(\underline{x}, 0), f_1(\underline{x}, y+1) = f_1(\underline{x}, y) + g(\underline{x}, y+1).$$

$$(2) f_2(\underline{x}, 0) = g(\underline{x}, 0), f_2(\underline{x}, y+1) = f_1(\underline{x}, y) \cdot g(\underline{x}, y+1).$$

□

Predicates. A predicate $P(x_1, \dots, x_n)$ of n variables is a statement concerning these variables which is either true or false. In our case, the variables stand for elements of \mathbb{N} . Such a predicate is determined by the set $\{\underline{x} \in \mathbb{N}^n \mid P(\underline{x}) \text{ is true}\}$ (and in formal approaches to set theory, would be identified with this set).

Recall that, if $A \subseteq \mathbb{N}^n$, the characteristic function of A is the function

$$\chi_A : \mathbb{N}^n \rightarrow \{0, 1\} \text{ defined by } \chi_A(\underline{x}) = \begin{cases} 1 & \text{if } \underline{x} \in A \\ 0 & \text{if } \underline{x} \notin A. \end{cases}$$

If P is a predicate, χ_P is defined to be χ_A , where $A = \{\underline{x} \in \mathbb{N}^n \mid P(\underline{x}) \text{ is true}\}$.

Definition. Let \mathcal{C} be a primitively recursively closed class. A subset A of \mathbb{N}^n is said to be in \mathcal{C} if $\chi_A \in \mathcal{C}$. A predicate P of n variables is in \mathcal{C} if $\{\underline{x} \in \mathbb{N}^n \mid P(\underline{x}) \text{ is true}\}$ is in \mathcal{C} .

This is a somewhat awkward notation since “in” does not mean “is a member of”. If \mathcal{C} is the class of primitive recursive functions, we shall say A (or P) is primitive recursive, rather than A (or P) is in \mathcal{C} . Similar terminology will be used with the class of recursive functions defined later.

In the next lemma, the notation of propositional logic is used, and is assumed to be familiar. (Recall that \wedge means “and”, \vee means “or” and \neg means “not”. Thus $P \vee Q$ is true, where P and Q are predicates, when either P is true, or Q is true, or both.)

Lemma 2.2. Let \mathcal{C} be a primitively recursively closed class. If $A, B \subseteq \mathbb{N}^n$ and A, B are in \mathcal{C} , then $A \cup B$, $A \cap B$ and $\mathbb{N}^n \setminus A$ are in \mathcal{C} . Consequently, if P, Q are predicates of n variables in \mathcal{C} , then $P \vee Q$, $P \wedge Q$ and $\neg P$ are in \mathcal{C} .

Proof.

$$\begin{aligned} \chi_{A \cup B}(\underline{x}) &= \chi_A(\underline{x}) \vee \chi_B(\underline{x}) \\ \chi_{A \cap B}(\underline{x}) &= \text{sg}(\chi_A(\underline{x}) + \chi_B(\underline{x})) \\ \chi_{\mathbb{N}^n \setminus A}(\underline{x}) &= 1 \dot{-} \chi_A(\underline{x}) \end{aligned}$$

□

We next note that some familiar predicates of two variables are primitive recursive, for example $x = y$ (meaning the predicate $P(x, y)$ defined by $P(x, y)$ is true if and only if $x = y$).

Lemma 2.3. The predicates $x = y$, $x \neq y$, $x \leq y$, $x < y$, $x \geq y$, $x > y$ are primitive recursive.

Proof. Referring to the examples of primitive recursive functions given above, note that

$$\chi_{\neq}(x, y) = \text{sg}(|x - y|), \chi_{<}(x, y) = \text{sg}(x \dot{-} y)$$

and then use Lemma 2.2. In a slightly strange-looking notation, $=$ is $\neg(\neq)$, \leq is $< \vee =$, \geq is $\neg(<)$, etc. \square

Bounded Quantifiers. These are quantifiers of the form $\exists y \leq z$ and $\forall y \leq z$, where y, z are variables representing elements of \mathbb{N} .

Lemma 2.4. *Let \mathcal{C} be a primitively recursively closed class. If P is a predicate of $n + 1$ variables in \mathcal{C} , then the predicates Q, R of $n + 1$ variables defined below are in \mathcal{C} .*

- (1) $Q(x_1, \dots, x_n, z)$ is true $\Leftrightarrow \exists y \leq z (P(x_1, \dots, x_n, y) \text{ is true})$;
- (2) $R(x_1, \dots, x_n, z)$ is true $\Leftrightarrow \forall y \leq z (P(x_1, \dots, x_n, y) \text{ is true})$.

Proof. (1) $\chi_Q(\underline{x}, z) = \text{sg} \left(\sum_{y=0}^z \chi_P(\underline{x}, y) \right)$;

(2) $\chi_R(\underline{x}, z) = \prod_{y=0}^z \chi_P(\underline{x}, y)$. Now use Lemma 2.1. \square

Bounded Minimisation. Let P be a predicate of $n + 1$ variables. Define $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ by

$$f(\underline{x}, z) = \begin{cases} \text{the least } y \leq z \text{ such that } P(\underline{x}, y) \text{ is true} & \text{if such a } y \text{ exists} \\ z + 1 & \text{otherwise.} \end{cases}$$

(Here $\underline{x} \in \mathbb{N}^n$.) The notation for this is $f(\underline{x}, z) = \mu y \leq z P(\underline{x}, y)$.

Lemma 2.5. *If \mathcal{C} is a primitively recursively closed class and P is in \mathcal{C} , then f (as just defined) is in \mathcal{C} .*

Proof. This follows from Lemma 2.1, since

$$f(\underline{x}, z) = \sum_{t=0}^z \prod_{y=0}^t \text{sg}(1 \dot{-} \chi_P(\underline{x}, y)).$$

\square

Note. If $g : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ is in \mathcal{C} , then defining $P(\underline{x}, z)$ to be true if and only if $g(\underline{x}, z) = 0$, P is in \mathcal{C} ($\chi_P(\underline{x}, z) = 1 \dot{-} \text{sg}(g(\underline{x}, z))$). Thus, if $f(\underline{x}, z) = \mu y \leq z (g(\underline{x}, y) = 0)$, then f is in \mathcal{C} . On the other hand, every predicate P can be expressed in this way, with $g(\underline{x}, z) = 1 \dot{-} \chi_P(\underline{x}, z)$.

Definition by Cases. Let $f_1, \dots, f_k : \mathbb{N}^n \rightarrow \mathbb{N}$ be in \mathcal{C} (a primitively recursively closed class) and let P_1, \dots, P_k be predicates in \mathcal{C} , of n variables. Suppose that for all $\underline{x} \in \mathbb{N}^n$, exactly one of $P_1(\underline{x}), \dots, P_k(\underline{x})$ is true. Define $f : \mathbb{N}^n \rightarrow \mathbb{N}$ by

$$f(\underline{x}) = f_i(x) \quad \text{if } P_i(\underline{x}) \text{ is true, for } \underline{x} \in \mathbb{N}^n.$$

Lemma 2.6. *If f is so defined, then f is in \mathcal{C} .*

Proof. Just note that $f(\underline{x}) = f_1(\underline{x})\chi_{P_1}(\underline{x}) + \cdots + f_k(\underline{x})\chi_{P_k}(\underline{x})$. □

Again, P_i can be given by: $P_i(\underline{x})$ is true if and only if $g_i(\underline{x}) = 0$, where g_i is in \mathcal{C} .

More Examples.

- (1) The predicate of two variables, “ x divides y ” (written $x|y$) is primitive recursive. For $x|y \Leftrightarrow \exists t \leq y (x.t = y)$. If $P(x, y, t)$ is the predicate $x.t = y$, then P is primitive recursive, as $\chi_P(x, y, t) = \chi_{=(x.t, y)}$.
- (2) The predicate of one variable, “ x is prime”, is primitive recursive, for

$$x \text{ is prime} \Leftrightarrow (\neg \exists y \leq x (1 < y \wedge y < x \wedge y|x)) \wedge (1 < x).$$

- (3) The function $p(n)$ = the n th prime is primitive recursive. Since p has to be defined on \mathbb{N} , we let $p(0) = 2$, $p(1) = 3$, etc., so in fact $p(n)$ is the n th odd prime for $n > 0$. To prove p is primitive recursive, note that

$$p(n+1) = \text{least } p \text{ such that } (p(n) < p \text{ and } p \text{ is prime})$$

and this value of p is less than or equal to $p(n)! + 1$, since none of $p(0), \dots, p(n)$ divide $p(n)! + 1$, but some prime does divide $p(n)! + 1$. Thus, if

$$f(x, y) = \mu p \leq y (x < p \wedge (p \text{ is prime}))$$

then f is primitive recursive, and so is $h(x) = f(x, x! + 1)$. Since $p(0) = 2$, $p(n+1) = h(p(n))$, p is primitive recursive. In future, we prefer to write p_n rather than $p(n)$ for this function.

- (4) Let $v(n, m)$ be the highest power of p_n dividing m . This does not make sense when $m = 0$, but we define v by

$$v(n, m) = \mu y \leq m (\neg (p_n^{y+1} | m))$$

so v is primitive recursive. This gives $v(n, 0) = 1$, which will not cause problems. If $p = p_n$, we define $\log_p : \mathbb{N} \rightarrow \mathbb{N}$ by $\log_p(m) = v(n, m)$, a primitive recursive function. Thus \log_p is essentially the p -adic valuation (except that $\log_p(0) = 1$), rather than the logarithm function encountered in analysis.

- (5) Define $\text{quo}(x, y) = \lfloor \frac{y}{x} \rfloor$ to be the quotient when y is divided by x . Then quo is primitive recursive. For $\text{quo}(x, 0) = 0$, and

$$\text{quo}(x, y+1) = \begin{cases} \text{quo}(x, y) + 1 & \text{if } y+1 = x(\text{quo}(x, y) + 1) \\ \text{quo}(x, y) & \text{otherwise.} \end{cases}$$

Thus, if we define

$$h(x, y, z) = \begin{cases} z+1 & \text{if } y+1 = x(z+1), \text{ i.e. } \underbrace{|(y+1) - x(z+1)|}_{P(x,y,z)} = 0 \\ z & \text{otherwise, i.e. if } \neg P(x, y, z) \text{ is true} \end{cases}$$

then h is primitive recursive by Lemma 2.6 (with $P_1 = P$, $P_2 = \neg P$). Since $\text{quo}(x, 0) = 0$ and $\text{quo}(x, y+1) = h(x, y, \text{quo}(x, y))$, it follows that quo is primitive recursive. Note that if $x = 0$, $\lfloor \frac{y}{x} \rfloor$ is undefined, but this definition gives $\text{quo}(0, y) = 0$ for all $y \in \mathbb{N}$.

(6) The remainder when y is divided by x

$$\text{rem}(x, y) = y - x \cdot \text{quo}(x, y) = y \dot{-} x \cdot \text{quo}(x, y)$$

is primitive recursive. Note that $\text{rem}(x, y) = y$ if $x = 0$, otherwise $0 \leq \text{rem}(x, y) < x$; also, $\text{rem}(1, y) = 0$.

Iteration. Let X be a set, $f : X \rightarrow X$ a partial function. The *iterate* of f is the partial function $F : X \times \mathbb{N} \rightarrow X$ defined by $F(x, 0) = x$, $F(x, n+1) = f(F(x, n))$ (so $F(x, n) = f^n(x)$ in the usual sense if f is total).

We have a notion of a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ being in a class \mathcal{C} . We can extend this to functions $f : \mathbb{N}^n \rightarrow \mathbb{N}^k$, by saying that f is in \mathcal{C} if the coordinate functions $\pi_{ik} \circ f$ are in \mathcal{C} for $1 \leq i \leq k$.

Definition. A class \mathcal{C} of functions is *closed under iteration* if, whenever $f : \mathbb{N}^n \rightarrow \mathbb{N}^n$ is in \mathcal{C} , then its iterate $F : \mathbb{N}^{n+1} \rightarrow \mathbb{N}^n$ is in \mathcal{C} .

One can show that if \mathcal{C} is primitively recursively closed, then \mathcal{C} is closed under iteration (see Exercise 6 at the end of this chapter, or just refer to [4, p. 40]). However, we are interested in a kind of converse.

Lemma 2.7. *Let \mathcal{C} be a class of functions which contains the initial functions and is closed under composition and iteration. Then \mathcal{C} is closed under primitive recursion.*

Proof. Let $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ be obtained from g, h by primitive recursion, where g, h are in \mathcal{C} . For $\underline{x} \in \mathbb{N}^n$, let $\varphi(\underline{x}, y, z) = (\underline{x}, y+1, h(\underline{x}, y, z))$ and let Φ be the iterate of φ . Then $\Phi(\underline{x}, 0, g(\underline{x}), y) = (\underline{x}, y, f(\underline{x}, y))$ (by induction on y). It is easy to see φ is in \mathcal{C} , so Φ is in \mathcal{C} . Since g is in \mathcal{C} and \mathcal{C} is closed under composition, it follows easily that f is in \mathcal{C} . \square

We now come to the more general classes of recursive and partial recursive functions. Their definition involves just one more way of constructing new functions, minimisation. Let $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ be a partial function. We can define a new function $g : \mathbb{N}^n \rightarrow \mathbb{N}$ by saying $g(\underline{x})$ is the least y such that $f(\underline{x}, y) = 0$. However, since f is partial, this needs some clarification, and the definition is as follows.

Definition. The function obtained from f by minimisation is the partial function $g : \mathbb{N}^n \rightarrow \mathbb{N}$ defined by

$$g(\underline{x}) = \begin{cases} r & \text{if } f(\underline{x}, r) = 0 \text{ and for } 0 \leq s \leq r, f(\underline{x}, s) \text{ is defined and not } 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

We write $g(\underline{x}) = \mu y (f(\underline{x}, y) = 0)$. By contrast with the bounded minimisation considered earlier, g may be partial even if f is total.

Definition. The function g is obtained from f by *regular minimisation* if, additionally, f is total and for all $\underline{x} \in \mathbb{N}^n$, there exists y such that $f(\underline{x}, y) = 0$. (The function g is then total.)

If f is computable in the informal sense described at the beginning of this section, then $g(\underline{x}) = \mu y (f(\underline{x}, y) = 0)$ is computable. To compute $g(\underline{x})$, take a procedure to compute f and use it to successively compute $f(\underline{x}, 0), f(\underline{x}, 1), f(\underline{x}, 2), \dots$ until a value r is reached with $f(\underline{x}, r) = 0$, then output r . This procedure will continue indefinitely if either a value s is reached with $f(\underline{x}, s)$ undefined (and none of $f(\underline{x}, 0), \dots, f(\underline{x}, s-1)$ is zero), or if there is no value of r such that $f(\underline{x}, r) = 0$. These are precisely the circumstances under which $g(\underline{x})$ is undefined.

A plausible way of defining g is to change the first clause as follows: $g(\underline{x}) = r$ if $f(\underline{x}, r) = 0$ and for $0 \leq s \leq r$, $f(\underline{x}, s)$ is either undefined or is defined and not equal to 0. However, the procedure just given will no longer work. If this clause applies and there is some $s < r$ with $f(\underline{x}, s)$ undefined, the procedure will continue indefinitely without outputting r . In fact, there are examples where, using this definition, one can argue that g is not computable (see the end of §2.4, p.32 in [4]). This is why we have not used this as the definition.

We are now ready to define the idea of recursive function.

Definition. The class of recursive functions is the smallest class \mathcal{C} of total functions which is primitively recursively closed and closed under regular minimisation. (That is, if f is in \mathcal{C} and g is obtained from f by regular minimisation, then g is in \mathcal{C} .)

Note that there is such a smallest class, namely the intersection of all such classes \mathcal{C} . As indicated earlier, a subset A of \mathbb{N}^n is called *recursive* if χ_A is recursive, and a predicate P of n variables is *recursive* if $\{\underline{x} \in \mathbb{N}^n \mid P(\underline{x}) \text{ is true}\}$ is recursive.

Thus the lemmas above concerning predicates in a primitively recursively class apply to recursive predicates.

The idea of a recursive subset of \mathbb{N}^n is the formal version of a *decidable* set. This is a set A for which there is a finite set of instructions for a procedure which, given $\underline{x} \in \mathbb{N}^n$, decides in finitely many steps whether or not $\underline{x} \in A$. Even with such a vague idea, it should be clear that A is decidable if and only if χ_A is computable.

Definition. The class of partial recursive functions is the smallest class of partial functions which contains the initial functions and is closed under composition, primitive recursion and minimisation (in what should be an obvious sense).

The class of partial recursive functions which are total is primitively recursively closed and closed under regular minimisation, so contains the class of recursive

functions. That is, a recursive function is partial recursive and total. The converse is true, but it is not obvious and will be proved later. Also, a primitive recursive function is recursive. We note some examples to show we have really extended the class of primitive recursive functions, and not all partial recursive functions are recursive.

Examples.

- (1) Let $f(x) = \mu y(x(y+1) = 0) = \begin{cases} 0 & \text{if } x = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$.

Clearly f is partial recursive but not total, so not recursive.

- (2) For examples of recursive functions which are not primitive recursive, see [4, §3.6]. A particularly interesting example is the function $A : \mathbb{N}^2 \rightarrow \mathbb{N}$ now generally known as Ackermann's function. It is a simplified version of Ackermann's original function, and is defined by

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

This is not a variant of primitive recursion, and A is not primitive recursive. But A is recursive, and it should be clear that A is computable, in the intuitive sense given at the beginning of the chapter. For proofs, see [5, §3.6.2].

Register Programs. Consider a machine having a number of registers, which are storage devices, each of which can store a non-negative integer. The machine can be given instructions to perform certain simple operations on registers. After finitely many steps, only finitely many registers are used, the others being clear (i.e. have 0 stored in them). However, there is no limit on the number that can be used. It is convenient to view the machine as having infinitely many registers, numbered $1, 2, 3, \dots$, where only finitely many have a non-zero entry.

x_1	x_2	x_3	\dots	x_n	0	0	\dots
1	2	3		n			

Figure 2.1

The register contents are described by an infinite sequence $\underline{x} = (x_1, x_2, x_3, \dots)$ of natural numbers, indexed by the positive integers, with $x_k = 0$ for all but finitely many values of k . Let Σ be the set of all such sequences.

Instructions are given to the machine by means of a *program*. We shall give a formal definition of a program, then indicate the intended meaning of the instructions.

Definition. A *register program* P is a finite sequence $\alpha_1, \dots, \alpha_r$ where each α_i has the form $i.\beta$, and β_i is an *instruction*, that is, one of

$$a_k, s_k, \text{STOP}, J_k(l, m)$$

where $k \geq 1$ and $1 \leq l, m \leq r$. We also require that α_r is *terminal*, i.e. of the form $r.\text{STOP}$.

We call i the *label* on α_i , and α_i is called a *line* of the program. The intended meaning of the instructions is as follows:

- a_k : add 1 to the contents of register k ;
- s_k : subtract one from the contents of register k , if this is not zero;
- $J_k(l, m)$: if register k is clear (i.e. contains 0), jump to instruction labelled l , otherwise to the instruction labelled m .

The instructions are executed in order unless a jump or STOP instruction is encountered. The STOP instruction means exactly what it says—when it is encountered no further instructions are carried out. Following the usual practice, the lines of a register program are written in a vertical list.

Example. Consider the program $O(k)$:

1. $J_k(4, 2)$
2. s_k
3. $J_k(4, 2)$
4. STOP

Starting at Line 1, if register k is clear we go to Instruction 4 and stop, otherwise go to Instruction 2 and subtract 1 from register k . Then we go to Instruction 3. If register k is now clear, we go to 4, otherwise back to 2. Thus, while register k is not clear, Instructions 2 and 3 are repeatedly executed until it is. That is, $O(k)$ clears the contents of register k .

Given n , it is easy to construct a register program using more than n registers. Thus the machine which runs all register programs must have infinitely many registers. An alternative approach is to give each register program its own machine, with finitely many registers, sufficient to run the program. This would no longer be possible if we considered more complicated programs with instructions of the form “if r is the contents of register k , do something related to register r ”.

We now give formal definitions of the effect that any register program P has on the registers of our machine.

Definition. A *configuration* of P is a pair (i, \underline{x}) , where i is a label and $\underline{x} \in \Sigma$. It is *terminal* if the line labelled i is terminal, i.e. is $i.\text{STOP}$.

(The interpretation is that \underline{x} represents the contents of the registers and the instruction of line i is about to be executed.) Given a non-terminal configuration (i, \underline{x}) , carrying out Instruction i will result in a new configuration, which is described in the following definition.

Definition. If (i, \underline{x}) is a non-terminal configuration, the configuration (j, \underline{y}) *yielded* by (i, \underline{x}) is defined by:

- (1) if line i has instruction a_k , then $j = i + 1, y_p = \begin{cases} x_p & \text{if } p \neq k \\ x_p + 1 & \text{if } p = k \end{cases}$
- (2) if line i has instruction s_k , then $j = i + 1, y_p = \begin{cases} x_p & \text{if } p \neq k \\ x_p \dot{-} 1 & \text{if } p = k \end{cases}$
- (3) if line i has instruction $J_k(l, m)$, then $\underline{y} = \underline{x}, j = \begin{cases} l & \text{if } x_k = 0 \\ m & \text{otherwise} \end{cases}$

Definition. The *computation* of P starting from $\underline{x} \in \Sigma$ is the finite or infinite sequence

$$(i_1, \underline{x}_1), (i_2, \underline{x}_2), \dots$$

where $i_1 = 1, \underline{x}_1 = \underline{x}$ and $(i_{q+1}, \underline{x}_{q+1})$ is the configuration yielded by (i_q, \underline{x}_q) , unless (i_q, \underline{x}_q) is the last term in the sequence, in which case it must be terminal.

This defines a partial function $\varphi_P : \Sigma \rightarrow \Sigma$:

$$\underline{x} \varphi_P = \begin{cases} \underline{y} & \text{if the computation of } P \text{ starting from } \underline{x} \text{ is a finite sequence} \\ & \text{whose last term is } (i, \underline{y}) \text{ for some } i \\ \text{undefined} & \text{otherwise} \end{cases}$$

(It is convenient to write φ_P on the right, as will be seen later.) For example, if $P = O(k)$,

$$\underline{x} \varphi_P = (x_1, \dots, x_{k-1}, 0, x_{k+1}, \dots)$$

Using φ_P , P determines a partial function $\mathbb{N}^n \rightarrow \mathbb{N}$, for every $n \geq 1$.

Definition. The partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is *computed* by the register program P if

$$f(x_1, \dots, x_n) = \begin{cases} y & \text{if } (x_1, \dots, x_n, 0, 0, \dots) \varphi_P = (y, \dots) \\ \text{undefined} & \text{if } (x_1, \dots, x_n, 0, 0, \dots) \varphi_P \text{ is undefined} \end{cases}$$

(In the first case, f is given the value in register 1, and the values in the other registers are irrelevant.)

Abacus Machines. These are not really machines, but are words meant to represent certain (well-structured) register programs. The alphabet is $\{a_k, s_k, (,)_k \mid k \geq 1\}$, which is infinite, including an infinite collection of indexed right parentheses, $)_1,)_2$, etc. To each abacus machine is associated a natural number (its *depth*) and there is a notion of *simple* abacus machine. The definition is by induction on depth.

- (1) a_k, s_k ($k \geq 1$) are the only simple abacus machines of depth 0.
- (2) The abacus machines of depth n are the words $M_1 \dots M_r$, where each M_i is a simple abacus machine of depth at most n , and some M_i has depth exactly n .
- (3) The simple abacus machines of depth $n + 1$ are the words $(M)_k$, where M is an abacus machine of depth n and $k \geq 1$.

An abacus machine is a set of instructions for operating on the registers, as follows.

a_k : add 1 to contents of register k ; s_k : subtract 1 from register k unless it contains 0.

$M_1 \dots M_r$: execute M_1, \dots, M_r in succession.

$(M)_k$ while $x_k \neq 0$ do M , where x_k is the contents of register k . (That is, check if $x_k \neq 0$ and if so, execute M . Do this repeatedly until $x_k = 0$.)

Note that the set of instructions corresponding to M may not terminate, for example $a_k(a_k)_k$ just keeps incrementing register k by 1. Here are some examples which carry out useful tasks.

Examples.

- (1) $Clear_k = (s_k)_k$ (clears the contents of register k).
- (2) $Descopy_{p,q} = Clear_q(s_p a_q)_p$ (copies contents of register p to register q and clears register p . This is short for “destructive copy” since the contents of register p are destroyed).
- (3) $Copy_{p,q,r} = Clear_q(s_p a_q a_r)_p (s_r a_p)_r$ (if register r is clear, copies register p to register q , leaving registers other than q unchanged).

Next we prove some results on the structure of an abacus machine.

- Lemma 2.8.** (1) *An abacus machine has the same number of left and right parentheses (all the letters $)_k$, $k \geq 1$ are regarded as right parentheses here).*
- (2) *A proper non-empty prefix of a simple abacus machine has more left than right parentheses (the proper non-empty prefixes of a word $u_1 \dots u_m$ are $u_1 \dots u_l$ where $1 \leq l < m$).*

Proof. We use induction on depth, when (1) becomes obvious. Clearly (2) holds for simple abacus machines of depth 0 (they have no proper non-empty prefixes). Suppose (2) holds for simple abacus machines of depth at most n , and let M be a simple abacus machine of depth $n+1$. Then $M = (M_1 \dots M_r)_k$, where each M_i is a simple abacus machine of depth at most n . The proper non-empty prefixes of M are $(M_1 \dots M_{i-1} M'_i)$, where M'_i is a prefix of M_i (possibly ϵ or M_i). By (1) and the induction hypothesis, $M_1, \dots, M_{i-1}, M'_i$ all have at least as many left as right parentheses, hence so does $M_1 \dots M_{i-1} M'_i$. Therefore $(M_1 \dots M_{i-1} M'_i)$ has more left than right parentheses. \square

- Lemma 2.9.** (1) *If a string S is an abacus machine, then there is exactly one value of r and one sequence of simple abacus machines M_1, \dots, M_r such that $S = M_1 \dots M_r$.*
- (2) *If S is a simple abacus machine, there is a unique k such that S is either a_k , s_k or $(M)_k$, where M is an abacus machine uniquely determined by S .*

Proof. (1) We can write $S = M_1 \dots M_r$ for some simple abacus machines M_1, \dots, M_r . By Lemma 2.8, M_1 is the shortest prefix of S (other than ϵ) having the same number of left and right parentheses. If $M_1 \neq S$, we can write $S = M_1 S'$ and similarly M_2 is the smallest prefix of S' having the same number of left and right parentheses. Continuing, this determines M_1, \dots, M_r (and r) uniquely.

(2) This is obvious since $)_k$ is the last letter of $(M)_k$, and $(M)_k = (M')_k$ implies $M = M'$ (deleting the first and last letters on each side). \square

An abacus machine M defines a partial function $\varphi_M : \Sigma \rightarrow \Sigma$, as follows.

- (1) $\underline{x} \varphi_{a_k} = \underline{y}$, where $y_i = \begin{cases} x_i & \text{if } i \neq k \\ x_i + 1 & \text{if } i = k \end{cases}$
- (2) $\underline{x} \varphi_{s_k} = \underline{y}$, where $y_i = \begin{cases} x_i & \text{if } i \neq k \\ x_i - 1 & \text{if } i = k \end{cases}$
- (3) If $M = M_1 \dots M_r$ (M_i simple) then $\underline{x} \varphi_M = \underline{x} \varphi_{M_1} \dots \varphi_{M_r}$.
- (4) If $M = (M')_k$, then $\underline{x} \varphi_M = \underline{x} \varphi_{M'}^t$, where $t \in \mathbb{N}$ is chosen as small as possible such that $(\underline{x} \varphi_{M'}^t)_k$ (the k th entry of $\underline{x} \varphi_{M'}^t$) is zero ($\underline{x} \varphi_M$ is undefined if no such t exists).

This defines φ_M by induction on the depth of M , using Lemma 2.9. (Having defined φ_M for M of depth at most n , (4) then defines φ_M for simple abacus machines of depth $n + 1$, then (3) defines it for all abacus machines of depth $n + 1$.) Incidentally, (3) is the reason φ_M is written on the right, so that the order of the M_i does not have to be reversed, and we write φ_P (where P is a register program) on the right for consistency. As with register programs, an abacus machine determines a function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ for each $n > 0$.

Definition. A partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is *computed* by the abacus machine M if: $f(x_1, \dots, x_n)$ is defined if and only if $(x_1, \dots, x_n, 0, 0, \dots) \varphi_M$ is in which case

$$(x_1, \dots, x_n, 0, 0, \dots) \varphi_M = (f(x_1, \dots, x_n), \dots).$$

As with the definition of computable by a register program, the entries other than the first in $(f(x_1, \dots, x_n), \dots)$ are irrelevant, but we show that they can all be taken to be 0. For this, the idea of registers used by an abacus machine is needed.

Definition. The registers *used* by an abacus machine M are those whose numbers appear as subscripts in the machine. Thus

- (1) a_k, s_k use only register k
- (2) $M_1 \dots M_r$ uses those registers used by M_i for at least one value of i
- (3) $(M)_k$ uses register k and the registers used by M .

Since only finitely many subscripts appear in an abacus machine, an abacus machine uses only finitely many registers.

Remark. If $\underline{x} \varphi_M = \underline{y}$ and register i is not used by M , then $x_i = y_i$. (This is easily proved by induction on depth.)

Lemma 2.10. If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is computed by an abacus machine, it is computed by an abacus machine M such that:

$f(x_1, \dots, x_n)$ is defined if and only if $(x_1, \dots, x_n, 0, 0, \dots) \varphi_M$ is, in which case

$$(x_1, \dots, x_n, 0, 0, \dots) \varphi_M = (f(x_1, \dots, x_n), 0, 0, \dots).$$

Proof. Let f be computed by M' . Choose m greater than or equal to the number of any register used by M' , and with $m \geq n$. Then $M = M' \text{Clear}_2 \dots \text{Clear}_m$ is the required machine. \square

The goal now is to show that register program computable, abacus machine computable and partial recursive are equivalent notions. The first step is the following, which implies that abacus machine computable functions are register program computable. The proof spells out the assertion that abacus machines are meant to represent certain register programs.

Lemma 2.11. *If M is an abacus machine, there is a register program P such that $\varphi_P = \varphi_M$ and the only STOP instruction of P is in the last line.*

Proof. (1) If $M = a_k$, take P to be $\begin{cases} 1.a_k \\ 2.\text{STOP} \end{cases}$ and if $M = s_k$, take P to be $\begin{cases} 1.s_k \\ 2.\text{STOP} \end{cases}$.

(2) Suppose $M = M_1 \dots M_r$, where there exist register programs P_i such that $\varphi_{P_i} = \varphi_{M_i}$ ($1 \leq i \leq r$) and P_i has only one STOP instruction. We show that there is a register program P with only one STOP instruction and $\varphi_P = \varphi_M$. For notational convenience, we treat only the case $r = 2$, leaving the modifications in the general case to the reader.

Let P_1 have labels $1, \dots, n$ and P_2 have labels $1, \dots, p$. Re-label the lines of P_2 as $n+1, \dots, n+p$ and replace any jump instructions $J_k(l, m)$ by $J_k(n+l, n+m)$, to get a sequence of lines P'_2 . Replace line n of P_1 by $n.J_1(n+1, n+1)$ (an unconditional jump to the line labelled $n+1$) to obtain P'_1 . Let P be the concatenation $P'_1 P'_2$; then P is a register program with only one STOP instruction, and $\varphi_P = \varphi_M$.

(3) Suppose $\varphi_{P'} = \varphi_{M'}$, where P' has r lines and one STOP instruction, and $k \geq 1$. We construct a register program P with one stop instruction and $\varphi_P = \varphi_{(M')^k}$. Increase all labels of P' by 1 and replace any jump instructions $J_q(l, m)$ by $J_q(l+1, m+1)$. Add a new first line, $1.J_k(r+2, 2)$, then remove the last line ($r+1.\text{STOP}$) and add two new lines: $\begin{cases} r+1.J_k(r+2, 2) \\ r+2.\text{STOP} \end{cases}$. This gives the required program P .

The lemma now follows by induction on the depth of the abacus machine M . \square

For example, if $M = \text{Clear}_k$, the program P with $\varphi_P = \varphi_M$ given by the proof is $O(k)$. We next show that partial recursive functions are abacus computable. Two technical lemmas about abacus computability are needed.

Remark. If $\underline{x}, \underline{y} \in \Sigma$ and $x_i = y_i$ for all i such that the abacus machine M uses register i , then $\underline{x} \varphi_M = \underline{y} \varphi_M$. This is easily proved by induction on the depth of M .

Lemma 2.12. *Let $f_1, \dots, f_r : \mathbb{N}^n \rightarrow \mathbb{N}$ be abacus computable and let $p \geq 0$ be an integer. Then there is an abacus machine N such that, for all $\underline{x} \in \Sigma$,*

$$\underline{x} \varphi_N = (x_1, \dots, x_n, x_{n+1}, \dots, x_{n+p}, f_1(\underline{x}), \dots, f_r(\underline{x}), \dots)$$

where $f_i(\underline{x})$ means $f_i(x_1, \dots, x_n)$.

Proof. Let the abacus machine M_i compute f_i ($1 \leq i \leq r$). Choose an integer m greater than the number of any register used by any of the M_i . Put $M'_i =$

$Clear_{n+1} \dots Clear_m M_i$. By the remark, for any $\underline{x} \in \Sigma$, $\underline{x} \varphi_{M'_i} = (f_i(x_1, \dots, x_n), \dots)$. Let M''_i be the machine obtained from M'_i by increasing every subscript of M'_i by q , where $q = p + r + n$. Let

$$K_i = Copy_{1,q+1,q+n+1} \dots Copy_{n,q+n,q+n+1} M''_i$$

Then for any $\underline{x} \in \Sigma$,

$$\underline{x} \varphi_{K_i} = (x_1, \dots, \underbrace{x_{n+p+r}}_q, f_i(x_1, \dots, x_n), \dots)$$

Now put $N_i = K_i Descopy_{q+1,n+p+i}$. Then $N = N_1 \dots N_r$ is the required machine. \square

Corollary 2.13. *Under the hypotheses of Lemma 2.12, there is an abacus machine M such that, for all $\underline{x} \in \Sigma$,*

$$\underline{x} \varphi_M = (f_1(x_1, \dots, x_n), \dots, f_r(x_1, \dots, x_n), x_{n+1}, \dots, x_{n+p}, \dots).$$

Proof. Let N be as in Lemma 2.12 and put $q = p + r + n$. Then

$$M = N Descopy_{n+1,q+1} \dots Descopy_{n+p,q+p} Descopy_{n+p+1,1} \dots Descopy_{n+p+r+p,r+p}$$

is the required machine. \square

Theorem 2.14. *Partial recursive functions are abacus computable.*

Proof. We show that the set of abacus computable functions contains the initial functions and is closed under composition, primitive recursion and minimisation. By definition, the class of partial recursive functions is then a subset, proving the theorem.

Now $Clear_1$ computes the zero function, a_1 the successor function, $Descopy_{k,1}$ ($k \neq 1$) computes π_{kn} and $a_1 s_1$ computes π_{1n} , so the initial functions are abacus computable.

Suppose $f_1, \dots, f_r : \mathbb{N}^n \rightarrow \mathbb{N}$ and $g : \mathbb{N}^r \rightarrow \mathbb{N}$ are abacus computable. By Cor. 2.13, there is an abacus machine M such that

$$(x_1, \dots, x_{n+1}, \dots) \varphi_M = (f_1(x_1, \dots, x_n), \dots, f_r(x_1, \dots, x_n), x_{n+1}, \dots).$$

Let g be computed by the abacus machine M' , and choose m greater than the number of any register used by M . Then

$$M Clear_{r+1} \dots Clear_m M'$$

computes $g \circ (f_1, \dots, f_r)$. Thus the set of abacus computable functions is closed under composition.

Let $f : \mathbb{N}^n \rightarrow \mathbb{N}^n$ be such that its coordinate functions $f_i = \pi_{in} \circ f$ are abacus computable for $1 \leq i \leq n$. We show that the iterate of f is abacus computable (meaning its coordinate functions are abacus computable). By Cor. 2.13, there is an abacus machine M such that

$$(x_1, \dots, x_{n+1}, \dots) \varphi_M = (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n), x_{n+1}, \dots).$$

Let $M' = (Ms_{n+1})_{n+1}$. Then

$$(x_1, \dots, x_{n+1}, \dots) \varphi_{M'} = (f_1^k(x_1, \dots, x_n), \dots, f_n^k(x_1, \dots, x_n), 0, \dots).$$

provided the right-hand side is defined, where $k = x_{n+1}$. Using M' and $M' \text{Descopy}_{i,1}$ ($2 \leq i \leq n$), we see that the iterate of f is abacus computable. By Lemma 2.7, the class of abacus computable functions is closed under primitive recursion.

Finally, let $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ be abacus computable. By Lemma 2.12, there is an abacus machine M such that for all $\underline{x} \in \Sigma$,

$$(x_1, \dots, x_{n+1}, \dots) \varphi_M = (x_1, \dots, x_{n+1}, f(x_1, \dots, x_{n+1}), \dots).$$

Let $M' = \text{Clear}_{n+1} M (a_{n+1} M)_{n+2} \text{Descopy}_{n+1,1}$. Then M' computes the function h given by $h(x_1, \dots, x_n) = \mu y (f(x_1, \dots, x_n, y) = 0)$. Thus the set of abacus computable functions is closed under minimisation, completing the proof. \square

The next result finishes the proof that abacus computable, register machine computable and partial recursive are equivalent.

Theorem 2.15. *If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is a partial function computed by a register program, then f is partial recursive.*

Proof. Let f be computed by the register program P with labels $1, \dots, r$. The mapping $(i, \underline{x}) \mapsto 2^i \prod_{m \geq 1} p_m^{x_m}$ is a one-to-one mapping from the set of configurations of P into \mathbb{N} , and $2^i \prod_{m \geq 1} p_m^{x_m}$ is called the *code* of (i, \underline{x}) . (Note that $g \in \mathbb{N}$ is a code if and only if $1 \leq \log_2 g \leq r$.) Define

$$\text{In}(x_1, \dots, x_n) = 2 \prod_{1 \leq m \leq n} p_m^{x_m}$$

(the code of $(1, (x_1, \dots, x_n, 0, 0, \dots))$) and

$$\text{Out}(g) = \log_3(g)$$

(the contents of register 1 if g is a code). Also define $\text{Next} : \mathbb{N} \rightarrow \mathbb{N}$ by:

$$\text{Next}(g) = \begin{cases} g, & \text{if } g \text{ is not a code, or is the code of a terminal configuration} \\ \text{code of the configuration yielded by } (i, \underline{x}), & \\ \text{where } g \text{ is the code of } (i, \underline{x}), & \text{otherwise} \end{cases}$$

In the second case, put $i = \log_2(g)$. Then

if line i of P is $i.a_k$, then $Next(g) = 2.g.p_k$

if line i of P is $i.s_k$, then $Next(g) = \begin{cases} 2.\text{quo}(p_k, g) & \text{if } \log_{p_k}(g) \neq 0 \\ 2.g & \text{if } \log_{p_k}(g) = 0 \end{cases}$

if line i of P is $i.J_k(l, m)$, then $Next(g) = \begin{cases} 2^m.\text{quo}(2^i, g) & \text{if } \log_{p_k}(g) \neq 0 \\ 2^l.\text{quo}(2^i, g) & \text{if } \log_{p_k}(g) = 0 \end{cases}$

Let $Comp$ be the iterate of $Next$. Finally, Let

$$Term(g) = \begin{cases} 1 & \text{if } g \text{ is the code of a terminal configuration} \\ 0 & \text{otherwise} \end{cases}$$

Then In , Out , $Next$, $Comp$ and $Term$ are primitive recursive (exercise).

Now, putting $\underline{x} = (x_1, \dots, x_n)$,

$$f(\underline{x}) = Out(Comp(In(\underline{x}), t))$$

for any t such that $Comp(In(\underline{x}), t)$ is the code of a terminal configuration, that is, such that $Term(Comp(In(\underline{x}), t)) = 1$ (and $f(\underline{x})$ is undefined if there is no such t). Put

$$\begin{aligned} F(\underline{x}, t) &= Out(Comp(In(\underline{x}), t)) \\ G(\underline{x}, t) &= 1 \dot{-} Term(Comp(In(\underline{x}), t)) \end{aligned}$$

Then F and G are primitive recursive, and

$$\begin{aligned} f(\underline{x}) &= F(\underline{x}, t) \text{ for any } t \text{ such that } G(\underline{x}, t) = 0 \\ &\text{(undefined if there is no such } t\text{).} \end{aligned}$$

Hence

$$f(\underline{x}) = F(\underline{x}, \mu t (G(\underline{x}, t) = 0)).$$

and it follows that f is partial recursive, being obtained from F and G by minimisation and composition. \square

Corollary 2.16. *For a partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$, the following are equivalent.*

- (1) f is partial recursive.
- (2) f is abacus computable.
- (3) f is computable by a register program.

Proof. (1) \Rightarrow (2) by Theorem 2.14, (2) \Rightarrow (3) by Lemma 2.11, (3) \Rightarrow (1) by Theorem 2.15. \square

We can now resolve a point from earlier in the chapter.

Corollary 2.17. *A partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is recursive if and only if it is partial recursive and total.*

Proof. It has already been noted that recursive implies partial recursive and total. If f is partial recursive, then it is computable by a register program (Cor. 2.16), and from the proof of Theorem 2.15, we can write $f(\underline{x}) = F(\underline{x}, \mu t(G(\underline{x}, t) = 0))$ for some primitive recursive functions F and G . If f is total, the minimisation must be regular, so f is recursive. \square

Computation of functions by Turing Machines. We show that the class of functions computable by a Turing machine coincides with the class of partial recursive functions. First, we have to specify how a TM computes a function, and this involves a special kind of TM.

Definition. A numerical TM is a deterministic TM $T = (Q, F, A, I, \tau, q_0)$ with $F = I = \emptyset$, $A = \{0, 1\}$ and $B = 0$ (blank symbol).

If $\underline{x} = (x_1, \dots, x_n) \in \mathbb{N}^n$, define $Tape(\underline{x})$ to be the tape description $01^{x_1}01^{x_2}0 \dots 01^{x_n}$. If T is a numerical TM, define $In_{T,n} : \mathbb{N}^n \rightarrow C$ (C is the set of configurations of T) by $In_{T,n}(\underline{x}) = (q_0, Tape(\underline{x}))$.

Definition. The partial function $\varphi_{T,n} : \mathbb{N}^n \rightarrow \mathbb{N}$ is defined by: if T , started on tape description $Tape(\underline{x})$ halts with the tape description $01^y = Tape(y)$ for some $y \in \mathbb{N}$ (i.e. the computation starting with $In_{T,n}(\underline{x})$, where $\underline{x} \in \mathbb{N}^n$, ends with a terminal configuration $(q, Tape(y))$), then $\varphi_{T,n}(\underline{x}) = y$. Otherwise, $\varphi_{T,n}(\underline{x})$ is undefined.

The partial function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is called *TM computable* if $f = \varphi_{T,n}$ for some numerical TM T .

It is convenient to modify T . Add two new states p, h and the transitions

$$\left. \begin{array}{l} qapaL \\ pahaR \\ hapaL \end{array} \right\} \text{ for all } (q, a) \in Q \times A \text{ such that no element of } \tau \text{ starts with } qa$$

$$\left. \begin{array}{l} pahaR \\ hapaL \end{array} \right\} \text{ for all } a \in A \text{ (i.e. } a = 0, 1).$$

Call the new machine T' , let $Q' = Q \cup \{p, h\}$ be its set of states and let C' be its set of configurations. Then T' remains deterministic and transitions have the form

$$qaN_{T'}(q, a)R_{T'}(q, a)D_{T'}(q, a)$$

(see p. 17, just before Lemma 1.10) and $N_{T'}, R_{T'}, D_{T'}$ are defined on $Q' \times A$. Also, after suitable renaming, we can assume that

$$Q' = \{0, 1, \dots, r-1\}, \quad h = 0, \quad p = 1, \quad L = 0, \quad R = 1.$$

Then $Q' \times A$ is a finite subset of \mathbb{N}^2 , and putting $N_{T'}(x, y) = R_{T'}(x, y) = D_{T'}(x, y) = 0$ for $(x, y) \in \mathbb{N}^2 \setminus (Q' \times A)$, $N_{T'}, R_{T'}, D_{T'}$ are primitive recursive functions $\mathbb{N}^2 \rightarrow \mathbb{N}$.

Let $\delta : C' \rightarrow C'$ be the transition function of T' , and let $\bar{\delta}$ be its iterate (these are total functions). If T has a computation ending with a terminal configuration $(q, Tape(y))$, then T' , after two more moves, can enter state h without altering the tape. The only moves are then to alternate between states p and h , alternately moving the tape right and left. Note that $In_{T,n} = In_{T',n}$ and so

$$\varphi_{T,n}(\underline{x}) = \begin{cases} y, & \text{for any } t \text{ such that } \bar{\delta}(In_{T,n}(\underline{x}), t) = (h, \underline{01}^y) \text{ for some } y \\ \text{undefined,} & \text{if there is no such } t \end{cases}$$

To show $\varphi_{T,n}$ is partial recursive, we need to code configurations by natural numbers. Define $Code : C' \rightarrow \mathbb{N}$ by $Code(q, a, \alpha, \beta) = 2^q 3^a 5^{\sigma(\alpha)} 7^{\sigma(\beta)}$ where

$$\begin{aligned} \sigma(\alpha) &= \alpha(0) + \alpha(1)2 + \alpha(2)2^2 + \dots \\ \sigma(\beta) &= \beta(0) + \beta(1)2 + \beta(2)2^2 + \dots \end{aligned}$$

so $Code$ is a $1-1$ function.

Lemma 2.18. *There is a primitive recursive function $Next : \mathbb{N} \rightarrow \mathbb{N}$ such that*

$$Next(Code(c)) = Code(\bar{\delta}(c))$$

for all $c \in C'$.

Proof. See Appendix A. □

Now let $Comp$ be the iterate of $Next$, so

$$Comp(Code(c), t) = Code(\bar{\delta}(c, t))$$

which follows by an easy induction on t . Also,

$$Code(h, \underline{01}^y) = 2^0 3^0 5^{1+2+2^2+\dots+2^{y-1}} 7^0 = 5^{2^y-1}$$

hence

$$\varphi_{T,n}(\underline{x}) = \log_2(1 + \log_5(Comp(Code(In_{T,n}(\underline{x})), t)))$$

for any t such that $Comp(Code(In_{T,n}(\underline{x})), t) = Code(h, \underline{01}^y)$ for some y (and is undefined if there is no such t). Define $\psi : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ by

$$\psi(\underline{x}, t) = Comp(Code(In_{T,n}(\underline{x})), t)$$

so ψ is primitive recursive, since $\underline{x} \mapsto Code(In_{T,n}(\underline{x}))$ is primitive recursive (the proof is left to the reader). Further, the predicate P defined by

$$P(\underline{x}, t) \text{ is true if and only if } \psi(\underline{x}, t) = Code(h, \underline{01}^y) = 5^{2^y-1} \text{ for some } y$$

is primitive recursive (again left as an exercise). Put

$$\begin{aligned} F(\underline{x}, t) &= \log_2(1 + \log_5(\psi(\underline{x}, t))) \\ G(\underline{x}, t) &= 1 \dot{-} \chi_P(\underline{x}, t) \end{aligned}$$

so F and G are primitive recursive. Then $\varphi_{T,n}(\underline{x}) = F(\underline{x}, t)$ for any t such that $G(\underline{x}, t) = 0$ and is undefined if there is no such t . In particular,

$$\varphi_{T,n}(\underline{x}) = F(\underline{x}, \mu t (G(\underline{x}, t) = 0))$$

is partial recursive. We have proved the following.

Theorem 2.19. *A TM computable function is partial recursive.*

□

We can take this further, by not only coding computations, but coding the TM's themselves. First, we need to order the transitions in a specific way. Given a linearly ordered set L , we can linearly order L^* . If $u = x_1 \dots x_m$, $v = y_1 \dots y_n \in L^*$, let $u < v$ if either $m < n$, or $m = n$ and there exists i such that $x_1 = y_1, \dots, x_{i-1} = y_{i-1}$ but $x_i < y_i$. This is called the *ShortLex* ordering on L^* . Restricted to the set of words of a fixed length, it is called the *lexicographic* ordering.

Now let T be a numerical TM, and modify it as above to obtain T' , so the transitions are words of length 5 in \mathbb{N}^* , and \mathbb{N} is linearly ordered. We can therefore order the transitions by the lexicographic ordering, then number them to respect this ordering, say $q_i a_i q'_i a'_i D_i$ ($1 \leq i \leq k$) (so this is the i th transition in the lexicographic ordering, and k is the number of transitions). Define

$$gn(T') = 2^k 3^{q_0} \prod_{i=1}^k p_{5i}^{q_i} p_{5i+1}^{a_i} p_{5i+2}^{q'_i} p_{5i+3}^{a'_i} p_{5i+4}^{D_i}$$

(gn stands for “Gödel numbering”, an idea discussed in the next chapter). Now define the following primitive recursive functions:

$$\begin{aligned} x(g, i) &= \log_{p_{5i}}(g), \quad y(g, i) = \log_{p_{5i+1}}(g) \\ k(g) &= \log_2(g) \\ j &= j(g, a, b) = \mu i \leq k(g) (x(g, i) = a \wedge y(g, i) = b) \\ N(g, a, b) &= \begin{cases} \log_{p_{5j+2}}(g) & \text{if } (\exists i \leq k) (x(g, i) = a \wedge y(g, i) = b) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Then if $g = gn(T')$, $N(g, a, b) = N_{T'}(a, b)$, where $N_{T'}$ is the function used in the proof of Lemma 2.18, defined just before Lemma 1.10. Similarly, we can define $R(g, a, b)$ and $D(g, a, b)$ (using $5j+3$, $5j+4$ instead of $5j+2$). Also, we define

$$In_n(g, \underline{x}) = (\log_3(g), Tape(\underline{x})), \text{ for } g \in \mathbb{N}, \underline{x} \in \mathbb{N}^n.$$

In the proof of Lemma 2.18 (see Appendix A), replace $R_{T'}(a, b)$, $N_{T'}(a, b)$, $D_{T'}(a, b)$ by $R(g, a, b)$, etc., to get a primitive recursive function $NEXT : \mathbb{N}^2 \rightarrow \mathbb{N}$ such that, if $g = gn(T')$, $NEXT(g, x) = Next(x)$. Define $COMP : \mathbb{N}^3 \rightarrow \mathbb{N}$ by

$$\begin{aligned} COMP(g, x, 0) &= x \\ COMP(g, x, t+1) &= NEXT(g, COMP(g, x, t)). \end{aligned}$$

Then $COMP$ is primitive recursive, and if $g = gn(T')$, $COMP(g, x, t) = Comp(x, t)$, where $Comp$ is the function in the proof of Theorem 2.19. This is proved by induction on t .

Put $\Psi(g, \underline{x}, t) = \text{COMP}(g, \text{Code}(\text{In}_n(g, \underline{x})), t)$ and let $P(g, \underline{x}, t)$ be the predicate

$$\Psi(g, \underline{x}, t) = 5^{2^y-1} \text{ for some } y.$$

Let

$$\begin{aligned} F(g, \underline{x}, t) &= \log_2(1 + \log_5(\Psi(g, \underline{x}, t))) \\ G(g, \underline{x}, t) &= 1 \dot{-} \chi_P(g, \underline{x}, t). \end{aligned}$$

Then F and G are primitive recursive, and if $g = gn(T')$, then from the proof of Theorem 2.19

$$\varphi_{T,n}(\underline{x}) = \begin{cases} F(g, \underline{x}, t) & \text{for any } t \text{ such that } G(g, \underline{x}, t) = 0 \\ \text{undefined} & \text{if no such } t \text{ exists} \end{cases}$$

We have now proved the following.

Theorem 2.20. *For each $n \geq 1$, there are primitive recursive functions $F, G : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ such that, for any TM computable function $f : \mathbb{N}^n \rightarrow \mathbb{N}$, there exists $g \in \mathbb{N}$ such that, for all $\underline{x} \in \mathbb{N}^n$, $f(\underline{x}) = F(g, \underline{x}, t)$ for any t such that $G(g, \underline{x}, t) = 0$ and $f(\underline{x})$ is undefined if no such t exists. In particular,*

$$f(\underline{x}) = F(g, \underline{x}, \mu t(G(g, \underline{x}, t) = 0)).$$

□

We shall prove that partial recursive functions are TM computable by showing that abacus computable implies TM computable. To do this, we need to construct some numerical TM's to perform specific tasks. There is quite a long list of them, but they provide insight into how TM's operate. A useful operation in their construction is the *product* of two TM's. This can only be defined when the first TM has a *halting state*.

Definition. Let T be any TM. A state h is called a *halting state* if, for any configuration $c = (q, a, \alpha, \beta)$, c is terminal if and only if $q = h$.

Products of TM's. Let T, T' be any TM's and assume T has a halting state. Rename the states so the halting state of T is also the initial state of T' , and T, T' have no other states in common. Also, assume T, T' have the same blank symbol. Define TT' to be the TM whose states and transitions are those of T and T' , and whose tape alphabet is the union of the tape alphabets of T and T' . The initial state and input alphabet are those of T , the final states are those of T' .

If T_1, \dots, T_r are TM's and T_1, \dots, T_{r-1} all have halting states, we define (recursively) $T_1 \dots T_r = (T_1 \dots T_{r-1})T_r$.

Some Numerical TM's

- (1) P_0 : this TM has set of states $Q = \{q_0, q, q'\}$ (where q_0 is the initial state) and four transitions

$$q_0 a q_0 R, \quad q a q' a L \quad (a = 0, 1).$$

P_i : has the same set of states, but transitions q_0aq_1R , $qaq'aL$.

(For $i = 0$ or 1 , P_i prints i on the scanned square and halts without moving the tape.)

- (2) R : this has $Q = \{q_0, q\}$ and transitions q_0aq_aR ($a = 0, 1$).
 L : has $Q = \{q_0, q\}$ and transitions q_0aq_aL ($a = 0, 1$).
 (R , respectively L , moves one square right (resp. left) and halts.)
- (3) R^* : $Q = \{q_0, q, q', h\}$, transitions

$$q_0aq_aR \ (a = 0, 1), \ q_1q_1R, \ q_0q'0R, \ q'ahaL \ (a = 0, 1).$$

(Moves to the first blank square to the right of the scanned square and halts.)

L^* : this is obtained from R^* by interchanging L and R in the transitions. (Moves to the first blank square to the left of the scanned square and halts.)

- (4) $Test$: this has $Q = \{q_0, p_0, p_1, p'_0, p'_1\}$ and transitions:

$$\begin{aligned} q_00p'_00R, \quad p'_0ap_0aL \quad (a = 0, 1) \\ q_01p'_11R, \quad p'_1ap_1aL \quad (a = 0, 1). \end{aligned}$$

($Test$ leaves the tape description unaltered, changes to state p_0 (respectively p_1) if 0 (resp. 1) is scanned initially, and halts.)

- (5) $Test\{T_0, T_1\}$: here T_0, T_1 are numerical TM's, with their states renamed so that they have no states in common, the initial state of T_i is p_i (the state of $Test$ for $i = 0, 1$, and $T_i, Test$ have only the state p_i in common. The states and transitions are those of T_0, T_1 and $Test$, and the initial state is that of $Test$.

(Started on a given tape description, this TM will follow the computation of T_0 or T_1 , according to whether a 0 or 1 is initially scanned.)

- (6) $Shiftright = P_1R^*LP_0R$ (started on the tape description $u001^x0v$, halts with the tape description $u01^x00v$, for any $u, v \in \{0, 1\}^*$).

$Shiftright = P_1L^*RP_0L$ (started on $u01^x00v$, halts with $u001^x0v$).

- (7) $Test_k = R^{*k-1}RTest\{L^{*k}, L^{*k}\}$. To describe the action of $Test_k$, define, for $\underline{x} \in \Sigma$

$$Tape(\underline{x}) = 01^{x_1}01^{x_2}01^{x_3}0 \dots$$

($Test_k$, started on $Tape(\underline{x})$, halts with the same tape description, but in a state p_0 if $x_k = 0$, and in a state p_1 if $x_k \neq 0$.)

This completes our list, and we can now use these TM's to show that abacus computable implies TM computable.

Definition. A numerical TM T with a halting state *simulates* an abacus machine M if, for all $\underline{x} \in \Sigma$, T , when started on the tape description $Tape(\underline{x})$, halts if and only if $\underline{x}\varphi_M$ is defined, in which case it halts with the tape description $Tape(\underline{x}\varphi_M)$.

Theorem 2.21. Any abacus machine M can be simulated by a numerical TM with a halting state.

Proof. The proof is by induction on the depth of M . If M is a_k , M is simulated by $Add_k = Shiftright^{k-1}P_1L^{*k}$. If M is s_k , then M is simulated by

$$Sub_k = R^{*k-1} RTest\{L^{*k}, T_k\}$$

where $T_k = P_0 LShiftright^{k-1} R$.

If $M = M_1 \dots M_r$ and T_i simulates M_i , then $T_1 \dots T_r$ simulates $M_1 \dots M_r$.

Suppose $M = (N)_k$ and T simulates N . Rename the states of T so its initial state is p_1 (a state of $Test_k$), its halting state is q_0 (the initial state of $Test_k$), but T and $Test_k$ have no other states in common. Let T' be the TM whose states and transitions are those of T and $Test_k$, with initial state q_0 . Then T' simulates M . This is left to the reader (the halting state of T' is the state p_0 of $Test_k$). \square

Corollary 2.22. *If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is abacus computable, there exists a numerical TM T with a halting state such that, started on the tape description $Tape(\underline{x})$ (where $\underline{x} \in \mathbb{N}^n$), T halts if and only if $f(\underline{x})$ is defined, in which case T halts with the tape description 01^y , where $y = f(\underline{x})$.*

A function is partial recursive \Leftrightarrow it is abacus computable \Leftrightarrow it is TM computable.

Proof. By Lemma 2.10, there is an abacus machine M such that $f(\underline{x})$ is defined if and only if $(x_1, \dots, x_n, 0, 0, \dots) \varphi_M$ is, for $x = (x_1, \dots, x_n) \in \mathbb{N}^n$, in which case

$$(x_1, \dots, x_n, 0, 0, \dots) \varphi_M = (f(\underline{x}), 0, 0, \dots).$$

By Theorem 2.21, there is a TM T which simulates M , and T is the required TM. Thus abacus computable implies TM computable, and the corollary follows by Cor. 2.16 and Theorem 2.19. \square

Our final result in this chapter makes use of this and Theorem 2.20.

Theorem 2.23 (Kleene Normal Form Theorem). *There exist primitive recursive functions $\varphi : \mathbb{N} \rightarrow \mathbb{N}$ and $\psi : \mathbb{N}^3 \rightarrow \mathbb{N}$ such that, if $f : \mathbb{N} \rightarrow \mathbb{N}$ is partial recursive, there exists $g \in \mathbb{N}$ such that*

$$f(x) = \varphi(\mu t(\psi(g, x, t) = 0)).$$

Proof. By Theorem 2.20 and Cor. 2.22, there are primitive recursive functions $F, G : \mathbb{N}^3 \rightarrow \mathbb{N}$ such that if $f : \mathbb{N} \rightarrow \mathbb{N}$ is partial recursive, there exists $g \in \mathbb{N}$ such that $f(x) = F(g, x, t)$ for any t such that $G(g, x, t) = 0$ (and $f(x)$ is undefined if no such t exists). Given f , choose such a number g .

Now put $\varphi = F \circ J_3^{-1}$ and

$$\psi(s, x, y) = GJ_3^{-1}(y) + |K(y) - s| + |KL(y) - x|$$

where J_3, K and L are as in Exercises (3) and (4) at the end of this chapter. Thus $J_3^{-1}(y) = (K(y), KL(y), LL(y))$, and φ, ψ are primitive recursive.

If $\psi(g, x, y) = 0$ then $K(y) = g, KL(y) = x$ and $G(g, x, t) = 0$, where $t = LL(y)$, so $f(x) = F(g, x, t) = \varphi(y)$.

Conversely, if $f(x)$ is defined, it is equal to $F(g, x, t)$ for some t with $G(g, x, t) = 0$. Put $y = J_3(g, x, t)$. Then $f(x) = \varphi(y)$ and $\psi(g, x, y) = 0$.

Thus $f(x)$ is defined if and only if there exists y with $\psi(g, x, y) = 0$, in which case $f(x) = \varphi(y)$ for any such y . In particular, $f(x) = \varphi(\mu t(\psi(g, x, t) = 0))$. \square

There are other ways of precisely defining computable functions, including several minor variants of register programs. The proof that computable functions are partial recursive often follows the method of proof used above for register program computable and TM computable. (Roughly, code computations by natural numbers, using primitive recursive functions.) The proof of the converse tends to use simulation (for example, of abacus machines by TM's).

For further reading on recursive function theory, see [29].

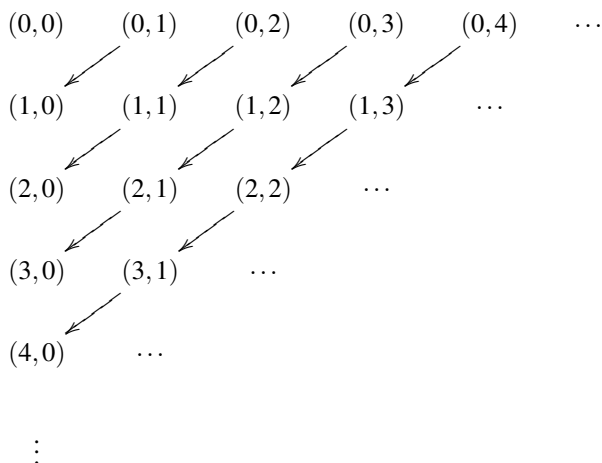
Exercises on Chapter 2

1. Show that the following functions are primitive recursive.

- (a) $f(x_1, \dots, x_n) = \max\{x_1, \dots, x_n\}$.
- (b) $f(x_1, \dots, x_n) = \min\{x_1, \dots, x_n\}$.
- (c) $f(x) =$ the number of primes less than or equal to x .

2. Show that the binary predicate RP, where $\text{RP}(x, y)$ means that x, y are relatively prime, is primitive recursive. Show that the function $\varphi : \mathbb{N} \rightarrow \mathbb{N}$, defined by $\varphi(x) =$ the number of positive integers less than or equal to x which are relatively prime to x , is primitive recursive.

3. We can define a bijection $J : \mathbb{N}^2 \rightarrow \mathbb{N}$ as follows. Write the elements of \mathbb{N}^2 as an infinite matrix:



then write the entries as an infinite sequence by successively moving along the diagonals from northeast to southwest, as indicated by the arrows, giving

$$(0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), (2,1), (3,0), (0,4), (1,3), \dots$$

Now there are $k+1$ pairs (m,n) with $m+n=k$, so the pair (m,n) occurs in the position $1+2+\dots+(m+n)+m$ in the sequence (where the first position is numbered 0). We therefore define $J(m,n) = \frac{1}{2}(m+n)(m+n+1) + m$.

- (a) Give a formal proof that J is bijective, and primitive recursive.
 - (b) Writing $J^{-1}(x) = (K(x), L(x))$, show that $K, L : \mathbb{N} \rightarrow \mathbb{N}$ are primitive recursive.
4. We can define bijections $J_n : \mathbb{N}^n \rightarrow \mathbb{N}$ for all n inductively by $J_1 = \pi_{11}$, $J_2 = J$ (the function in the previous exercises), $J_3(x_1, x_2, x_3) = J(x_1, J(x_2, x_3))$, and in general $J_{n+1}(x_1, \dots, x_{n+1}) = J(x_1, J_n(x_2, \dots, x_{n+1}))$. It follows easily by induction on n that J_n is primitive recursive for all n .
- Show that $J_3^{-1}(r) = (K(r), KL(r), LL(r))$, and that for all n , J_n^{-1} is primitive recursive (meaning its coordinate functions are primitive recursive).
5. If $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is a partial function with finite domain, show that f is partial recursive.
6. If $f : \mathbb{N}^n \rightarrow \mathbb{N}^n$ is in \mathcal{C} (a primitive recursively closed class), prove that its iterate $F : \mathbb{N}^{n+1} \rightarrow \mathbb{N}^n$ is in \mathcal{C} . (Hint: this is easy for $n=1$; in the general case, consider the iterate of $J_n \circ f \circ J_n^{-1}$.)
7. If $h : \mathbb{N} \rightarrow \mathbb{N}$ is primitive recursive, show that $\varphi : \mathbb{N}^3 \rightarrow \mathbb{N}$ defined by $\varphi(x, t, r) = h^{t \dot{-} r}(x)$ is primitive recursive.
8. Suppose $h, k : \mathbb{N} \rightarrow \mathbb{N}$ and $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ are primitive recursive, and $g : \mathbb{N}^2 \rightarrow \mathbb{N}$ is defined by

$$\begin{aligned} g(x, 0) &= k(x) \\ g(x, t+1) &= f(x, g(h(x), t)). \end{aligned}$$

Prove that g is primitive recursive.

(Hint: without mentioning g , give a definition of a function $G(x, t, r)$ by primitive recursion, such that $G(x, t, r) = g(h^{t-r}(x), t)$ for $t \geq r$. Using this definition, show that for $t \geq r$, $G(x, t+1, r) = G(h(x), t, r)$. Then put $g(x, t) = G(x, t, t)$ and show g is given by the equations above.)

9. Construct a numerical TM T_1 which, started on the tape description $u01^a001^c$, halts with the tape description $u01^a01^c\underline{0}$, for any $u \in \{0, 1\}^*$ and natural numbers a, c .

10. Construct a numerical TM T_2 which, started on the tape description $u01^a01^b\underline{0}1^c$, halts with the tape description $u01^{a+1}01^{b-1}\underline{0}1^{c+1}$, provided $b > 0$, for any $u \in \{0, 1\}^*$ and natural numbers a, b, c .
11. Construct a numerical TM T_3 which, started on the tape description $u01^a01^b\underline{0}$, halts with the tape description $u01^{a+b}01^b\underline{0}$, for any $u \in \{0, 1\}^*$ and natural numbers a, b .
12. Construct a numerical TM T_4 which, started on the tape description $u01^a01^b\underline{0}$, halts with the tape description $u01^{a+b}\underline{0}$, for any $u \in \{0, 1\}^*$ and natural numbers a, b .
13. Given a positive integer k , construct a numerical TM T_5 which, started on the tape description $u01^a01^b\underline{0}$, halts with the tape description $u01^{a+kb}\underline{0}$, for any $u \in \{0, 1\}^*$ and natural numbers a, b .
14. Given a positive integer k , construct a numerical TM T_6 which, started on the tape description $01^{x_1}01^{x_2} \dots 01^{x_n}\underline{0}$, halts with the tape description

$$\underline{0}1^{x_1+x_2k+x_3k^2+\dots+x_nk^{n-1}}$$

for any $n, x_1, \dots, x_n > 0$, and which, when started on a blank tape, halts on a blank tape (i.e. it works when $n = 0$ as well). (Your machine should have a halting state.)

(Hint: in **9–14**, use machines already constructed and products of TM's. In some cases, you may need to identify the initial state of one machine with a state of another machine.)

A Course in Formal Languages, Automata and Groups

Chiswell, I.M.

2009, IX, 157 p. 30 illus., Softcover

ISBN: 978-1-84800-939-4