Algorithms and Computability

Wayne Richter

March 7, 2021

Contents

Preliminaries

Definition 0.0.1.

$$\mathbb{N} = \{0, 1, 2, \ldots, n \ldots\} \text{is the set of } \textit{non-negative integers}; \tag{0.1}$$

$$\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots - n, n, \ldots\} \text{is the set of } \textit{integers}; \tag{0.2}$$

$$\mathbb{Q} = \{\frac{m}{n} : m, n \in \mathbb{Z} \;\&\; n \neq 0\} \text{is the set of } \textit{rational numbers}; \tag{0.3}$$

$$\mathbb{R} \text{ is the set of } \textit{real numbers}. \tag{0.4}$$

Set Notation and Functions

Suppose that $P(x)$ is a mathematical statement about x. $\{x : P(x)\}$ is the set of objects x such that $P(x)$ is true. For example, suppose that $P(x)$ is the statement: $x \in \mathbb{N} \;\&\; (x \text{ is divisible by } 2)$. In this case,

$$\{x : x \in \mathbb{N} \;\&\; (x \text{ is divisible by } 2)\}$$

is the set of those non-negative integers which are even. This same set may also be written as

$$\{x \in \mathbb{N} : x \text{ is divisible by } 2\} .$$

Quantifier abbreviations are often useful here. For example,

$$\{x \in \mathbb{N} : \exists y \in \mathbb{N}(x = 2 \times y)\}$$

is also the set of even non-negative integers. We also have sets whose members are ordered pairs of objects. One example is

$$\{(x, y) : x, y \in \mathbb{R} \;\&\; y < x^2\} .$$

A member of this set is an ordered pair of real numbers. (What does this set look like if we draw its picture on the x, y coordinate system?)

Definition 0.0.2. Let A and B be two sets.

i) $A \cup B = \{x : x \in A \text{ or } x \in B\}$; (the *union* of A and B)

ii) $A \cap B = \{x : x \in A \text{ and } x \in B\}$; (the *intersection* of A and B).

iii) $A \smallsetminus B = \{x : x \in A \ \& \ x \notin B\}$;

iv) $A \times B = \{(x, y) : x \in A \ \& \ y \in B\}$ (the *cartesian product* of A and B) is the set of ordered pairs whose first coordinate is in A and whose second coordinate is in B;

Similarly,

$$A \times B \times C = \{(x, y, z) : x \in A \ \& \ y \in B \ \& \ z \in C\}$$

v) $A \subseteq B$ if every member of A is a member of B (A is a *subset* of B).

vi) $A \subset B$ if $A \subseteq B$ but $A \neq B$ (A is a *proper subset* of B).

vii) If A_n is a set for each $n \in \mathbb{N}$ then $\bigcup\{A_n : n \in \mathbb{N}\}$ (the *union* of the sets A_n) is the set:

$$\bigcup\{A_n : n \in \mathbb{N}\} = \{x : \exists n \in \mathbb{N}\ x \in A_n\}\ .$$

An object x belongs to this union iff x belongs to at least one of the sets A_n. Note that $\{A_n : n \in \mathbb{N}\}$ is a set each of whose members is a set, i.e. it is a set of sets. More generally:

viii) Suppose X is a set of sets. Then

$$\bigcup X = \{x : \exists y \in X\ [x \in y]\}$$

is the union of the sets in X.

Definition 0.0.3 (Informal Definition)**.** A *function* f consists of

i) A set called the *domain* of f, written $\mathrm{dom}(f)$, and

ii) ii) a "correspondence" which associates with each member x of $\mathrm{dom}(f)$ a unique *value* $f(x)$. [1]

[1]This is not a good mathematical definition since the word 'correspondence' has not been defined.

Suppose f is a function. In the expression $f(x)$, x is called the *argument*, and $f(x)$ is called the *value* of the function f at argument x. The set

$$\{f(x) : x \in \operatorname{dom}(f)\}$$

of values of the function f is called the *range* of f and is written $\operatorname{ran}(f)$ (or sometimes $\operatorname{rng}(f)$).

Suppose f is a function. The *graph of* f is the set of pairs

$$\{(x, f(x)) : x \in \operatorname{dom}(f)\} .$$

Note that from the graph of f we can determine both $\operatorname{dom}(f)$ and $\operatorname{ran}(f)$. Members of the domain of f are the first coordinates of pairs belonging to the graph of f, and members of the range of f are second coordinates of pairs belonging to f. Since the graph of f uniquely determines f, it is customary to identify a function with its graph. A precise definition of 'function' can then be given as follows:

Definition 0.0.4. A function is a set of ordered pairs f such that:

$$\forall x\, \forall y\, \forall z\ [(x, y) \in f\ \&\ (x, z) \in f \Longrightarrow y = z] .$$

If f is a function, then

$$\operatorname{dom}(f) = \{x : \exists y\ (x, y) \in f\} ; \tag{0.5}$$
$$\operatorname{ran}(f) = \{y : \exists x\ (x, y) \in f\} . \tag{0.6}$$

Note that the statement $f(x) = y$ is equivalent to the statement $(x, y) \in f$.

Definition 0.0.5. Suppose f is a function.

i) f is *one-to-one* (or 1-1) if for all $x \in \operatorname{dom}(f)$,

$$f(x) = f(y) \Longrightarrow x = y .$$

ii) $f\colon A \to B$ (f *maps* A *into* B) means $A = \operatorname{dom}(f)$ and $\operatorname{ran}(f) \subseteq B$;

iii) $A \twoheadrightarrow B$ (f maps A *onto* B) means $A = \operatorname{dom}(f)$ and $\operatorname{ran}(f) = B$;

iv) $f\colon \subseteq A \to B$ means $\operatorname{dom}(f) \subseteq A$ and $\operatorname{ran}(f) \subseteq B$.

v) v) $f : A \xrightarrow{1-1} B$ means $f\colon A \to B$ and f is one-to-one.

Functions of two or more variables are actually special cases of the above definition. A function of two variables is a function f whose domain is a set of ordered pairs. For example the function $+ : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ (the addition function on $\mathbb{N}$) is really the set $\{((x, y), z) : x, y \in \mathbb{N} \;\&\; x + y = z\}$. A member of the domain of this function $+$ is a pair (x, y) where x and y are in $\mathbb{N}$.

Let A be a subset of some set U.[2] The *characteristic function of* A (with respect to the set U) is the function C_A, where[3] for all $x \in U$,

$$\mathrm{C}_A(x) = \begin{cases} 1, & \text{if } x \in A\,; \\ 0, & \text{otherwise.} \end{cases}$$

One-To-One Correspondence

Definition 0.0.6. Let A and B be two sets. A *one-to-one correspondence* (often written as $1 - 1$) between A and B is a function f such that $f : A \overset{1-1}{\twoheadrightarrow} B$

Example 0.0.7. Let

$$f = \{(3, \sqrt{2}), (-\pi, 0), (6, -2/3)\}\ .$$

Then:

(a) f is function (check the definition!)

(b) $\mathrm{dom}(f) = \{3, -\pi, 6\}$, and $\mathrm{ran}(f) = \{\sqrt{2}, 0, -2/3\}$.

(c) f is 1-1.

It follows that f is a 1-1 correspondence between $A = \{3, -\pi, 6\}$ and $B = \{\sqrt{2}, 0, -2/3\}$. Note that both A and B have 3 members. In general, if A is a set with n members (where $n \in \mathbb{N}$), and there exists a $1 - 1$ correspondence between A and B, then B will also have n members.

Definition 0.0.8. The set A is

a) *empty* if A has no members (The empty set is denoted by $\emptyset$);

[2]Neither A nor U need by subsets of $\mathbb{N}$ or indeed have any connection with $\mathbb{N}$ whatsoever.

[3]The characteristic function is sometimes defined with values 0 and 1 interchanged.

b) *finite* if for some $n \in \mathbb{N}$, A has n members;[4]

c) *infinite* if A is not finite;

d) *enumerable* if there exists a 1-1 correspondence between $\mathbb{N}$ and A;[5]

e) *countable* if A is finite or enumerable;

f) *uncountable* if A is not countable.

Remark 0.0.9.

1) If A is enumerable there is a *listing* $a_0, a_1, \ldots, a_n, \ldots$ *without repetitions* of the members of A (the subscripts are the members of $\mathbb{N}$). There are of course many different such listings of the same enumerable set, and each such listing might be very complicated.

2) Conversely, if there is a listing $a_0, a_1, \ldots, a_n, \ldots$ without repetitions of the members of A, then A is enumerable. (The one-to-one correspondence between $\mathbb{N}$ and A is given by the function f, where $f(n) = a_n$.)

3) If A is countable and non-empty, then there is a listing $a_0, a_1, \ldots, a_n, \ldots$ of the members of A (*with* repetitions if A is finite).

4) Conversely, if there is a listing $a_0, \ldots, a_n, \ldots$ (perhaps with repetitions) of A, then A is countable.

Proposition 0.0.10.

1) If A is enumerable and $B \subseteq A$ then B is countable.

2) If A is finite and $B \subseteq A$, then B is finite.

3) If B is infinite and $B \subseteq A$, then A is infinite.

Theorem 0.0.11.

a) If A and B are both countable, then each of $A \cup B$, $A \cap B$, and $A \times B$ is countable.

b) If each of $A_1, \ldots, A_k$ is countable then $A_1 \times \cdots \times A_k$ is countable.

---

[4] So the empty set $\emptyset$ is finite.

[5] Enumerable sets are also called *denumerable*.

Theorem 0.0.12. *$\mathbb{Q}$ is enumerable.*

Theorem 0.0.13. *If each of the sets $A_0, A_1, \ldots, A_n, \ldots$, $n \in \mathbb{N}$, is countable, then $\bigcup\{A_n : n \in \mathbb{N}\}$ is countable.*

Let A be an arbitrary set. We look at the set of all finite sequences from A. For example, if $a, b, c \in A$, then (a, b, c) is a sequence from A of length three. We also find it convenient to consider the *empty* sequence (the sequence of length 0) as being a finite sequence.

Theorem 0.0.14. *0.12. If $A \neq \emptyset$ is countable then the set of all finite sequences from A is enumerable.*

If A and B are finite sets, where A has n members and B has m members, then B is 'bigger' then A iff $m > n$. The following definition gives us an appropriate definition of 'bigger' which works for both finite and infinite sets.

Definition 0.0.15.

i) $A \preceq B$ if there is some function f such that $f : A \xrightarrow{1-1} B$; that is, $A \preceq B$ if there is a 1-1 correspondence between A and some subset of B.

ii) $A \prec B$ if $A \preceq B$, but *not* $B \preceq A$.

iii) $A \equiv B$ if $A \preceq B$ and $B \preceq A$.

We may think of:

a) $A \preceq B$ as a way of saying 'B is at least as big as A' (or 'A is not bigger than B').

b) $A \prec B$ as a way of saying 'B is bigger than A.'

c) $A \equiv B$ as a way of saying 'A and B are the same size.'

Exercise 0.0.16. 0.14. Show that if B is countable and $A \preceq B$ then A is countable.

There is another reasonable way of saying two sets are the same size: One might say that the sets A and B are the same size if there is a one-to-one correspondence between A and B. Is this equivalent to saying $A \equiv B$? Fortunately, we have the following result. We do not give a proof here. However, the proof is not *too* hard and you are encouraged to try to show why it is true.

Theorem 0.0.17 ((Cantor-Bernstein))**.** *$A \equiv B$ iff there is a one-to-one correspondence between A and B.*

It is quite possible that $A \equiv B$ even though A is a *proper* subset of B. The following is an example.

Example 0.0.18. Let $E = \{x \in \mathbb{N} : x \text{ is even}\}$. Then $E \equiv \mathbb{N}$.

Question 0.0.19. Is there a 'largest' set?

Definition 0.0.20. For any set A,

$$\mathcal{P}(A) = \{X : X \subseteq A\} .$$

$\mathcal{P}(A)$ is called the *power set* of A.

Exercise 0.0.21 ((Counting Subsets))**.** We count the number of subsets of a given finite set. Note that every set is a subset of itself; that is, for every set A, we have $A \subseteq A$.[6]

a) List all the subsets of:

   1) $\emptyset$ (the set with no members);
   2) A, where $A = \{a\}$ is a set with one member;
   3) B, where $B = \{a, b\}$ is a set with two members (i.e. $a \neq b$);
   4) C, where $C = \{a, b, c\}$ is a set with three members;

b) b) Let A be a set of objects, and b be an object not in A. Suppose A has n subsets. How many subsets has $A \cup \{b\}$? (*Hint:* How many subsets of $A \cup \{b\}$ are there which have b as a member? How may subsets of $A \cup \{b\}$ are there which do *not* have b as a member?)

c) Show by induction: For each $n \in \mathbb{N}$, if A has n members, then $\mathcal{P}(A)$ has 2^n members.

Theorem 0.0.22 ((Cantor))**.** *For every set A,*

$$A \prec \mathcal{P}(A) .$$

[6]So every set has at least one subset.

Corollary 0.0.23. *0.20.*

i) For every A there is a B such that $A \prec B$.
ii) $\mathcal{P}(\mathbb{N})$ is uncountable.

The following notation is convenient.

Definition 0.0.24. For any sets A and B,

$$^{A}B \text{ is the set of all functions } f \text{ such that } f\colon A \longrightarrow B.$$

Theorem 0.0.25. *$^{\mathbb{N}}\mathbb{N}$ is uncountable.*

The idea behind the proof of Cantor's Theorem will be used in this course a number of times in different contexts. It is called a *diagonal argument*. In order to understand the idea behind the proof, we look at the proof in a simpler setting, namely that of ???.

Proof. We give a proof by contradiction. Suppose that $\mathcal{P}(\mathbb{N})$ is countable. Since each of $\{0\}, \{1\}, \ldots, \{n\}, \ldots$ is in $\mathcal{P}(\mathbb{N})$, $\mathcal{P}(\mathbb{N})$ must be infinite, and hence enumerable. Thus there is a listing $A_0, A_1, \ldots, A_n, \ldots$ of the members of $\mathcal{P}(\mathbb{N})$. contradiction will be found by defining a subset B of $\mathbb{N}$ which is different from each of the sets A_0, A_1,. . .. Let C_0 be the characteristic function of the set A_0, C_1 be the characteristic function of A_1, and in general, C_n be the characteristic function of A_n. For $i, j \in \mathbb{N}$, let $c_{i,j} = C_i(j)$. So, for each i and j,

$$c_{i,j} = \begin{cases} 1, & \text{if } j \in A_i; \\ 0, & \text{otherwise.} \end{cases}$$

Think of the numbers $c_{i,j}$ as being arranged in an infinite square with the number $c_{i,j}$ in the $i + 1$=st row and $j + 1$-st column. So the numbers $c_{n,n}$ appear on a diagonal of the infinite square (the diagonal running from the upper left towards the lower right). Now let

$$d(n) = \begin{cases} 1, & \text{if } c_{n,n} = 0; \\ 0, & \text{if } c_{n,n} = 1. \end{cases}$$

d is called a *diagonal* function since it changes the values of the numbers on the diagonal of the square. Let

$$B = \{n : d(n) = 1\} .$$

Then $B \subseteq \mathbb{N}$, i.e. $B \in \mathcal{P}(\mathbb{N})$, and for all n,

$$\begin{aligned} nn \in B &\iff d(n) = 1 && (0.7)\\ &\iff c_{n,n} = 0 && (0.8)\\ &\iff n \notin A_n\ . && (0.9)\end{aligned}$$

It follows that B is different from each of the sets A_0, A_1,. . .. For suppose that for some n_0, $B = A_{n_0}$. This gives the contradiction:

$$\begin{aligned} n_0 \in A_{n_0} &\iff n_0 \in B && (0.10)\\ &\iff n_0 \notin A_{n_0}\ . && (0.11)\end{aligned}$$

□

Exercise 0.0.26. Let A be an enumerable set. By definition, there is a function

$$f : \mathbb{N} \overset{1-1}{\twoheadrightarrow} A \qquad (0.12)$$

But there is not a unique such function f. Let $\mathcal{F}$ be the set of all functions which satisfy (0.12). Is $\mathcal{F}$ countable? Explain why.

Relations on a set

Definition 0.0.27. Let A be a set.

a) A subset of $A \times A$ is a 2-*ary* (*binary*) *relation on* A.

b) A subset of $A \times A \times A$ is a 3-*ary* (*ternary*) *relation on* A.

c) More generally, a subset of $\underbrace{A \times \cdots \times A}_{n}$ is an *n-ary relation on* A.

Notation 0.0.28. *0.24. If R is an n-ary relation on A, then* $R(a_1, \dots, a_n)$ *means* $(a_1, \dots, a_n) \in R$. *If R is a binary relation on A, we sometimes write aRb to mean* $R(a, b)$.

Example 0.0.29. The sets

$$\begin{aligned} &\{(m, n) : m \in \mathbb{N}\ \&\ n \in \mathbb{N}\ \&\ m < n\} && (0.13)\\ &\{(m, n) : m \in \mathbb{N}\ \&\ n \in \mathbb{N}\ \&\ m > n\} && (0.14)\\ &\{(m, n) : m \in \mathbb{N}\ \&\ n \in \mathbb{N}\ \&\ m = n\} && (0.15)\end{aligned}$$

are binary relations on $\mathbb{N}$ called respectively the *less than* relation on $\mathbb{N}$, the *greater than* relation on $\mathbb{N}$, and the *equality* (or identity) relation on $\mathbb{N}$, and are abbreviated, respectively as $<^{\mathbb{N}}$, $>^{\mathbb{N}}$, and $=^{\mathbb{N}}$. Thus, for example,

$$a <^{\mathbb{N}} b \iff <^{\mathbb{N}} (a,b) \iff (a,b) \in \mathbb{N} \times \mathbb{N} \;\&\; a < b \;.$$

Chapter 1

The Informal Notion of Algorithms

The symbols m and n always stand for members of $\mathbb{N} = \{0, 1, 2, \ldots, n, \ldots\}$.

1.1 An Informal Definition of Algorithm

An *algorithm*[1] is a *finite* ordered list of instructions. When the instructions are executed it is called *running the algorithm*. Sometimes the algorithm will have a blank space to be filled in with a name of an integer.[2] When supplied with an integer n, we execute the instructions (which may depend upon the given integer n). This is called *running the algorithm on input n*. Sometimes the algorithm will 'halt' with an 'output' such as **yes** or **no**. Sometimes the algorithm will not halt, in which case there will be no output, or perhaps it will halt but give no output. Since a machine which does not halt gives no output, these are the only three possibilities. So when we say "the algorithm halts with output **yes**" this is the same as saying that "the algorithm gives output **yes**."

The notion of 'instruction' above is rather vague. You can think of it as meaning that, if you wanted to, it would be possible to write the instructions in your favorite programming language.[3] Our understanding of what constitutes an algorithm will become clearer as we proceed.

We shall use letters like $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ to stand for algorithms.

[1]Sometimes called an *effective procedure* or *mechanical procedure*.

[2]There might be more than one blank space, each of which is to be filled in with an integer.

[3]This isn't completely accurate, but it will do for now.

1.2 Algorithms with output yes or no

We now look at a special kind of algorithm: an *algorithm for determining membership in a set*. Such an algorithm has a blank space □ for an input, and two possible outputs **yes** and **no**. Let $A \subseteq \mathbb{N}$. Such an algorithm is an *algorithm for determining membership in the set* A if for each n, if the algorithm is run on input n, it will halt after a finite number of steps with output **yes** if $n \in A$, and halt with output **no** if $n \notin A$.

Example 1.2.1. Let $E = \{n \in \mathbb{N} : n \text{ is even}\}$. Is there an algorithm for determining membership in E?
If you have such an algorithm $\mathcal{A}$ it should have the following property: If you are 'handed' an arbitrary $n \in \mathbb{N}$ as input, then by executing the instructions which comprise the algorithm $\mathcal{A}$, after a finite number of steps the procedure will halt with output either **yes** or **no**, depending upon whether $n \in E$ or $n \notin E$. Note that one must specify the manner in which the integer n is 'handed' to you. You would actually be given a *name* for n, expressed presumably in some base. For example, it is very easy to describe such an algorithm for this particular set E if you are handed a name for n in base ten (and even easier if n is given in base two).

The following is one way of writing an algorithm $\mathcal{A}$ for determining membership in E when the input is given in base ten.

```
If the rightmost digit of □ is 0, 2, 4, 6, or 8, give output yes.
If not, give output no.
```

When a name in base ten for an integer n is placed in □ one can run the algorithm on input n.

Rather than actually using blank spaces □, it is slightly more readable to describe the algorithm as follows:

```
   On input n:
If the rightmost digit of n is 0, 2, 4, 6, or 8 give output yes.
If not, give output no.
```

For our purposes it is quite acceptable to write algorithms in this informal manner.[4] If you are a computer scientist you may want to write the algorithm in

[4]We have not specified how the output is given and the form is not particularly important. Perhaps there is a little man with a long white beard who shouts out the appropriate answer of **yes**, or **n**o. We shall assume, however, that the output is printed.

a style which is closer to that of your favorite programming language, but this is not at all necessary and sometimes just makes the algorithm more difficult to read.

Definition 1.2.2 (Informal)**.** Let A be a subset of $\mathbb{N}$. A is *effectively decidable* if there is an algorithm for determining membership in A.

Note that in the above definition, 'effectively decidable' was defined in terms of 'algorithm,' but we have not given a precise definition of an algorithm. For this reason, we have called this an *informal definition*. We are thinking of an algorithm as an informal, pre-mathematical notion.

Example 1.2.3.

a) $\mathbb{N}$ is effectively decidable. (`On input` n`: Give output` **`yes.`**)

b) $\emptyset$ is effectively decidable. (`On input` n`: Give output` **`no.`**)

c) Every finite subset of $\mathbb{N}$ is effectively decidable.

To see why (c) is true, suppose A is finite; say $A = \{a_0, a_1, \ldots, a_k\}$. `On input` n`: if` n `appears in the list` $a_0, \ldots, a_k$`, give output` **`yes.`** `If it doesn't, give output` **`no.`**

Question 1.2.4. 1.4. Let $A \subseteq \mathbb{N}$. Is A effectively decidable? Does the answer depend upon A?[5]

Theorem 1.2.5 (Informal)**.** *If A and B are effectively decidable subsets of $\mathbb{N}$, then so are $\mathbb{N} \setminus A$, $A \cup B$, and $A \cap B$. Therefore, the collection of effectively decidable subsets of $\mathbb{N}$ is closed under the operations of complementation, finite union, and finite intersection.*

Proof. for $A \cap B$. Let $\mathcal{A}$ be an algorithm for determining membership in A, and $\mathcal{B}$ be an algorithm for determining membership in B. The following algorithm C determines membership in $A \cap B$.

`On input` n`: Run` $\mathcal{A}$ `on input` n`, and simultaneously`[6] `run` $\mathcal{B}$ `on input` n`. If` $\mathcal{A}$ `gives output` **`yes`** `and` $\mathcal{B}$ `gives output` **`yes`** `then`

[5] A *Question* is something that you should think very hard about (preferably between 1 a.m. and 3 a.m.), but not necessarily be able to answer. Often the answer will come later. An *Exercise* is something that you should work out in detail.

[6] Or subsequently.

```
give output yes.
Otherwise give output no.
```
□

Exercise 1.2.6. Prove Theorem 1.2.5 for the cases $\mathbb{N} \setminus A$ and $A \cup B$.

The following famous problem is an example where the set in question is not a set of integers.

Example 1.2.7 (Hilbert's Tenth Problem)**.** Consider polynomials with integer coefficients (and any number of variables). For example, $3x^2 + 5xy - 2z^4 + 3$ is such a polynomial. A *diophantine equation* is an equation of the form $p = 0$, where p is such a polynomial. Two questions:

a) Is there an algorithm for determining whether or not diophantine equations have integer (positive or negative) solutions?

b) For every diophantine equation $p = 0$ is there an algorithm for determining whether or not the equation $p = 0$ has an integer solution?

Note that (a) asks if a statement of the form $\exists\mathcal{A}\ \forall D\ \ldots$ is true, whereas (b) asks if the statement $\forall D\ \exists\mathcal{A}\ \ldots$ is true. The answer to (b) is clearly **yes**. The answer to (a) is known as Hilbert's Tenth Problem. In 1900, Hilbert gave a list of important questions he could not answer. This was the tenth problem on his list. It was finally answered in 1970. Note that in 1900 when Hilbert asked this question, there was no precise *mathematical* definition of what was means by an algorithm. But the *informal* notion of algorithm was clear enough to make sense.

1.3 Algorithms with integer output: effectively computable functions

Until now we have considered algorithms that on any input n gave either an output of **yes** or an output of **no**. But if we think of an algorithm as simply a kind of finite list of instructions, it is easy to imagine algorithms with other kinds of output. For example, the following algorithm gives an integer as output.

```
On input $n$:  Give output $n+1$.
```

The following example shows it is even possible that for certain inputs an algorithm $\mathcal{A}$ gives no output at all.

```
On input n:  If n is even give output n.
```

Thus on input 4, $\mathcal{A}$ gives output 4, but on input 3, $\mathcal{A}$ gives no output.

For the rest of this section we deal only with subsets of $\mathbb{N}$ (or perhaps subsets of $\mathbb{N} \times \mathbb{N}$, etc.), and functions (of one or more variables) whose arguments are in $\mathbb{N}$, and whose values are in $\mathbb{N}$. Such functions are called *partial number-theoretic functions*. It is convenient to have terminology which enables us to distinguish between those functions which are defined for all arguments and those which are not.

Definition 1.3.1. a) A partial number-theoretic function is *total* if it is defined for all number arguments.

b) $f(n) \downarrow$ means $n \in \text{dom}(f)$, i.e. $f(n)$ is defined.

c) $f(n) \uparrow$ means $n \notin \text{dom}(f)$, i.e. $f(n)$ is undefined.

Thus a partial number-theoretic function of one variable is a function $\subseteq f : \mathbb{N} \to \mathbb{N}$. This function is total if $\text{dom}(f) = \mathbb{N}$ (and then we write $f : \mathbb{N} \to \mathbb{N}$). Similarly, a partial number-theoretic function of two variables is a function $\subseteq f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. This function is total if $\text{dom}(f) = \mathbb{N} \times \mathbb{N}$ and then we write $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Note that every total function is also partial.

Definition 1.3.2 ((Informal))**.** A partial function $\subseteq f : \mathbb{N} \to \mathbb{N}$ is *effectively computable*[7] if there is an algorithm $\mathcal{A}$ such that on input n:

a) if $n \in \text{dom}(f)$ then the algorithm $\mathcal{A}$ gives output $f(n)$.

b) if $n \notin \text{dom}(f)$ then $\mathcal{A}$ give no output.

In this case we say that $\mathcal{A}$ is an *algorithm for computing* f.

Note that if f is a *total* effectively computable function then there is an algorithm $\mathcal{A}$ which, for each n, on input n gives output $f(n)$.

[7] other terminology sometimes used is *mechanically calculable, algorithmically calculable*, or *effectively calculable*

Example 1.3.3. It is clear that the following total functions are effectively computable:

a) $f(n) = 0$ for all n (On input n: Give output 0.)

b) $f(n) = 1$ for all n (On input n: Give output 1.)

c) Every constant function (a) and (b) are special cases of this case.).

d) $f(n) = n$ for all n. The *identity function on* $\mathbb{N}$. (On input n: Give output n.)

We can also talk about effectively computable total functions of two or more variables. For example, a number-theoretic total function of two variables is effectively computable if there is an algorithm $\mathcal{A}$ such that for each m and n,

on input the ordered pair (m, n) the algorithm $\mathcal{A}$ gives output $f(m, n)$. The addition and multiplication functions $+$ and $\times$ are effectively computable total functions of two variables. The algorithms for computing $+$ and $\times$ (when inputs are given in base ten) are the algorithms for addition and multiplication that you learned in third(?) grade.

Theorem 1.3.4 (Informal)**.** *For any set $A \subseteq \mathbb{N}$, the characteristic function C_A is effectively computable iff the set A is effectively decidable.*

Remark 1.3.5. *Caution!* Let

$$f(n) = \begin{cases} 0, & \text{if in the decimal expansion of } \pi \text{ the consecutive} \\ & \text{sequence 123456789 appears;} \\ 1, & \text{otherwise.} \end{cases}$$

Is f effectively computable?

Convention. We shall use *effectively computable* to mean *total effectively computable* and *partial effectively computable* to refer to a function which might be either partial or total.

1.4 Other Forms of Output

Since an algorithm is a finite list of instructions, one can imagine an algorithm $\mathcal{A}$ whose output is another algorithm $\mathcal{B}$. More generally, suppose that for each

$n \in \mathbb{N}$, $\mathcal{B}_n$ is an algorithm. It is then conceivable that there is an algorithm $\mathcal{A}$ which on input n prints out the algorithm $\mathcal{B}_n$. Is it possible that $\mathcal{B}_n = \mathcal{A}$ for some n? That is, is it possible that for some input n, the algorithm $\mathcal{A}$ prints out itself?

1.5 Effectively enumerable sets

An algorithm for determining membership in a subset A of $\mathbb{N}$ gives, for each input n, an output of either **yes** or **no**. The following describes an algorithm used for a different purpose: listing members of a set.

Definition 1.5.1 (Informal)**.** Let $A \subseteq \mathbb{N}$. An *algorithm for listing the members of A* is an algorithm that, as it 'runs,' *lists* (i.e. *prints*, or *enumerates*), one at a time, numbers $a_0, a_1, \ldots, a_n, \ldots$ in such a way that

a) $a_n \in A$ for all $n \in \mathbb{N}$ (i.e. every number listed belongs to A);

b) If $a \in A$, then $a = a_n$ for some n (i.e. every member of A will turn up at some finite stage in the listing).

Note that an algorithm for listing the members of a set A has no blank space for an input. Repetitions in the listing are permitted. If the set A is finite, the algorithm will either terminate after listing a finite set of numbers, or it will keep running forever, printing repetitions (or perhaps printing nothing at all). If the set A is infinite, the algorithm will never stop 'running.' At any finite time it will have printed out a finite number of members of A (perhaps with some repetitions). And for every member a of A, the element a will turn up eventually in the listing at some finite time.

Example 1.5.2. We describe an algorithm (i.e. list of instructions) that lists the members of the set E of even nonnegative integers. The following is one way of describing such an algorithm:

Step 1 `print` 0;

Step 2 `print` 2;

In general, at

Step n `print` $2(n-1)$.

This description tells what to do at each step in the process of executing the instructions. However, an algorithm is supposed to be a *finite* list of instructions, and this looks like a separate instruction for each n. We can rephrase it in a more direct form as follows:

```
(1) Set $n = 0$ and go to (2);
(2) Print $n$ and go to (3);
(3) Replace $n$ by $n+2$ and with this 'new' $n$ return to (2).
```

This is a list of three instructions that accomplishes the same thing as the earlier description. Note that executing this list of three instructions still takes an infinite number of steps. We shall frequently describe algorithms in the first form, realizing that they can be rewritten explicitly as a finite list of instructions in the second form.

Definition 1.5.3 ((Informal))**.** Let $A \subseteq \mathbb{N}$. A is *effectively enumerable* if there is an algorithm for listing the members of A.

Note that the empty set $\emptyset$ is effectively enumerable. An algorithm $\mathcal{A}$ that lists $\emptyset$ is an algorithm that never prints anything.

Remark 1.5.4. We have introduced three informal concepts: *effectively decidable* and *effectively enumerable* for sets, and *effectively computable* for functions. There is a close connection between these three concepts.

Proposition 1.5.5 (Informal)**.** *If A is effectively decidable, then A is effectively enumerable.*

Question 1.5.6 ((Informal))**.** Is the converse of 1.5.5 true? That is, if A is effectively enumerable is A effectively decidable?

Proposition 1.5.7 ((Informal))**.**

a) Let $f : \mathbb{N} \longrightarrow \mathbb{N}$. If f is an effectively computable function, then $ran(f)$ is an effectively enumerable set.

b) If $A \neq \emptyset$ is an effectively enumerable subset of $\mathbb{N}$, then there is an effectively computable function f with $ran(f) = A$.

Combining (a) and (b) we have:

c) Let A be a non-empty subset of $\mathbb{N}$. A is effectively enumerable iff there is an effectively computable function whose range is equal to A.

Exercise 1.5.8. Let f be a strictly increasing, effectively computable function of one variable.[8] Show that the range of f is effectively decidable.

Hint: Suppose $\mathcal{A}$ is an algorithm for computing f. How do you describe an algorithm $\mathcal{B}$ for deciding membership in the range of f?

Exercise 1.5.9. Let f be an effectively computable function whose range is infinite. Show that there is an effectively computable, one-to-one, function g such that $\operatorname{ran}(g) = \operatorname{ran}(f)$.

Exercise 1.5.10. Show that if A and $N \setminus A$ are both effectively enumerable, then A is effectively decidable. It then follows from 1.5.5 that for a subset A of $\mathbb{N}$: *A is effectively decidable iff both A and $\mathbb{N} \setminus A$ are effectively enumerable.*

Exercise 1.5.11. Let $f(n) = n^2 - 6n$. The following describes an algorithm $\mathcal{A}$. On input n: If $n > 0$, compute the integers $f(n), f(n+1), f(n+2), \ldots$. In general, at step k compute $f(n+k-1)$. If a number k is reached such that $f(n) > f(n+k)$, then give output n.

a) What does $\mathcal{A}$ do on input 0?

b) What does $\mathcal{A}$ do on input 1?

c) What does $\mathcal{A}$ do on input 6?

Definition 1.5.12. a) A *binary relation* on $\mathbb{N}$ is a subset of $\mathbb{N} \times \mathbb{N}$.

b) More generally, an *n-ary relation on* $\mathbb{N}$ is a subset of $\mathbb{N}^n$.

Although we have been talking about effectively decidable and effectively enumerable subsets of $\mathbb{N}$, we can also talk about effectively decidable and effectively enumerable subsets of $\mathbb{N} \times \mathbb{N}$, $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$, etc. For example, suppose A is a binary relation on $\mathbb{N}$.

a) A is *effectively decidable* if there is an algorithm $\mathcal{A}$ such that given as input the ordered pair (m, n), the output of $\mathcal{A}$ will be **yes** if $(m, n) \in A$, and **no** if $(m, n) \notin A$.

b) A is *effectively enumerable* if there is an algorithm $\mathcal{A}$ that lists all pairs (m, n) belonging to A, in a manner similar to (a) and (b) of ???

[8] The function f is strictly increasing if for all m and n, we have $m < n \Rightarrow f(m) < f(n)$.

Example 1.5.13. The following binary relations on $\mathbb{N}$ are effectively decidable.

a) $<= \{(m,n) : m < n\}$;

b) $>= \{(m,n) : m > n\}$;

c) $== \{(m,n) : m = n\}$.

Theorem 1.5.14 (Informal)**.** *There are only enumerably many algorithms that can be written in any given language.*[9]

Corollary 1.5.15.

a) There are only enumerably many partial effectively computable functions. Since there are uncountably many total number-theoretic functions, there must exist total number-theoretic functions that are not effectively computable.

b) There are only enumerably many effectively enumerable subsets of $\mathbb{N}$. Therefore, since there are uncountably many subsets of $\mathbb{N}$, there must exist subsets of $\mathbb{N}$ that are not effectively enumerable.

By 1.5.15, there exists an enumeration $f_0, f_1, f_2, \ldots$ of all effectively computable functions of one variable. (There are many such enumerations of course.) Can this enumeration be effective in some sense? That is, is there an algorithm that on input n will print out an algorithm for computing the function f_n? The answer is no. To see this, let $g(n,m) = f_n(m)$. Then g is not an effectively computable function (of two variables). For suppose g is effectively computable. We get a contradiction as follows: Let $h(n) = g(n,n) + 1$. If g is effectively computable then so is h, but for all i, $h \neq f_i$. and hence is not effectively computable. We can express this in two ways as follows:

Theorem 1.5.16 (Informal)**.**

a) Let g be an effectively computable function of two variables. For each n, let f_n be the function of one variable: $f_n(m) = g(n,m)$. Then each of the functions in the list $f_0, f_1, \ldots, f_n, \ldots$ is effectively computable, but there is an effectively computable function that does not appear in the list.

b) If $f_0, \ldots, f_n, \ldots$ is a listing of all effectively computable functions of one variable, and g is the function of two variables defined by: $g(n,m) = f_n(m)$, then g is not effectively computable.

[9]We are assuming that the set of symbols in the language is enumerable.

The Halting Problem

Let $\mathcal{A}$ be an algorithm with, say, outputs **yes** and **no**. It is easy to define a modification $\mathcal{B}$ of $\mathcal{A}$ that behaves just like $\mathcal{A}$, except that whenever $\mathcal{A}$ halts with output **no**, $\mathcal{B}$ goes into an 'infinite loop' and never halts. Thus $\mathcal{B}$ halts with output **yes** if $\mathcal{A}$ halts with output **yes**, and $\mathcal{B}$ does not halt if $\mathcal{A}$ halts with output ***no***. One way of doing this would be to replace in $\mathcal{A}$ each instruction of the form: give output **no**, by the instructions:

(1) `set` $n = 0$ `and go to (2)`

(2) `replace` n `by` $n + 1$ `and go to (1)`

Now, suppose someone hands you an algorithm $\mathcal{A}$ (with inputs taken to be members of $\mathbb{N}$) and an input n. How can you tell whether or not $\mathcal{A}$ will halt on this input? It would be nice if there were an algorithm for doing this. Now for a particular n, say $n = 3$, it either halts or it doesn't. Perhaps with some effort you can decide if it does halt. But then what about input $n = 4$, or $n = 17$? It would be nice to have another algorithm $\mathcal{B}$ that does the following: on input n, if $\mathcal{A}$ halts on input n, then $\mathcal{B}$ halts with output **yes**. If $\mathcal{A}$ does not halt with input n, then $\mathcal{B}$ halts with output **no**.

We know that the set of algorithms that can be written in a given language is an enumerable set. Let $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n, \ldots$ be a listing of all algorithms written in this language that have just one integer input. Does there exist an algorithm $\mathcal{H}$ that on any given input (m, n) will halt with output **yes** if $\mathcal{A}_m$ halts on input n, and will halt with output **no** if $\mathcal{A}_m$ does not halt with input n?

Theorem 1.5.17 (Informal version of the Halting Problem)**.** *Let $\mathcal{A}_0, \ldots, \mathcal{A}_n, \ldots$ be a listing of all algorithms, with space for just one input, that can be written in the English language*[10] *(plus some math symbols). There does does not exist an algorithm $\mathcal{H}$ that on input the pair (m, n), halts with output* ***yes*** *if the algorithm $\mathcal{A}_m$ halts on input n, and halts with output* ***no*** *if $\mathcal{A}_m$ does not halt on input n.*

Proof. Suppose that $\mathcal{H}$ is such an algorithm. Let C be the following algorithm. On input n: If $\mathcal{H}$ halts on input (n, n) with output **no**, then C halts with output **yes**; if $\mathcal{H}$ halts on input (n, n) with output **yes**, then C does not halt. Now

[10] or Chinese!

C is an algorithm with just one integer input, and so it appears[11] in the list $\mathcal{A}_0, \ldots, \mathcal{A}_n, \ldots$. Say $C = \mathcal{A}_k$. Does C halt on input k?

C halts on input k iff

$\mathcal{H}$ halts on input (k, k) with output **no** iff

$\mathcal{A}_k$ does not halt on input k iff

C does not halt on input k (since $C = \mathcal{A}_k$).

It follows that there is no such algorithm $\mathcal{H}$. □

We shall return to this later in a more formal setting. This result, in a more formal setting, is described by saying that the halting problem is unsolvable. 'Unsolvable' in this context simply refers to the nonexistence of an algorithm. ???

Exercise 1.5.18. Suppose that R is an effectively enumerable subset of $\mathbb{N}$, and let

$$P(n) \iff \forall x < n\ R(x)\ .$$

Is P effectively enumerable? Explain why.

Suppose that Q is a binary relation on $\mathbb{N}$. Think of $\mathbb{N} \times \mathbb{N}$ as an (x, y)-coordinate system, that is as being the points in an infinite square with the origin $(0, 0)$ in the lower left corner. Then Q is a set of points in this coordinate system. From Q we can form the set $\{n : \exists m\ Q(n, m)\}$. This[12] is the *projection of Q on the 1st coordinate*. Similarly, $\{m : \exists n\ Q(n, m)$ is the *projection of Q on the 2nd coordinate*.

Exercise 1.5.19. Let $Q \subseteq \mathbb{N} \times \mathbb{N}$ be effectively enumerable, and P be defined by

$$P(n) \iff \exists m\, Q(n, m)\ .$$

Show that P is effectively enumerable.

[11] We are assuming that the language in which the algorithms are written is sufficiently powerful so that the algorithm C can also be written in this language.

[12] $Q(n, m)$ means $(n, m) \in Q$. More generally, for a k-ary relation R, $R(x_1, \ldots, x_k)$ means $(x_1, \ldots, x_k) \in R$.

Question 1.5.20. Suppose that $Q \subseteq \mathbb{N} \times \mathbb{N}$ is effectively decidable, and

$$P(n) \iff \exists m\, Q(n,m) \ .$$

Is P effectively decidable?

Exercise 1.5.21 (Selection)**.** Suppose that $Q \subseteq \mathbb{N} \times \mathbb{N}$ is effectively enumerable. Show that there is a partial effectively computable function f such that for all x,

a) $f(x) \downarrow$ iff $\exists y\ Q(x,y)$, and

b) if $\exists y\ Q(x,y)$ then $Q(x, f(x))$.

Exercise 1.5.22. Determine if the following statements are true, or false.

a) If A and B are effectively enumerable subsets of $\mathbb{N}$, then so are $A \cup B$ and $A \cap B$.

b) If each of $A_0, \ldots, A_n, \ldots$ is an effectively enumerable subset of $\mathbb{N}$, then so is $\bigcup\{A_n : n \in \mathbb{N}\}$.

c) If each of $A_0, \ldots, A_n, \ldots$ is an effectively enumerable subset of $\mathbb{N}$, then so is $\bigcap\{A_n : n \in \mathbb{N}\}$.

d) If $A \cup B$ is an effectively enumerable subset of $\mathbb{N}$, then both A and B are effectively enumerable.

e) If $\text{ran}(f)$ is effectively enumerable, then f is partial effectively computable.

f) If $\text{ran}(f)$ is finite, then f is partial effectively computable.

Exercise 1.5.23. Let $A_0, A_1, \ldots, A_n, \ldots$ be a listing of *all* the effectively decidable subsets of $\mathbb{N}$.

a) Let B be the binary relation on $\mathbb{N}$ defined by:

$$(m,n) \in B \iff m \in A_n \ .$$

Show that B is not effectively decidable.

b) Show that there does not exist an algorithm $\mathcal{A}$ that on input n, prints out an algorithm for determining membership in A_n.

Exercise 1.5.24. Let $A_0, \ldots, A_n, \ldots$ be a listing of *all* the effectively enumerable subsets of $\mathbb{N}$. Let B be the binary relation on $\mathbb{N}$ defined by:

$$(m, n) \in B \iff m \in A_n \ .$$

Try to use the method you used to solve Exercise 1.5.23 (a) to show that B is not effectively enumerable. Are you successful? Explain why.

Exercise 1.5.25. Suppose:

a) $A_0, \ldots, A_n, \ldots$ is a listing of all effectively enumerable subsets of $\mathbb{N}$, and

b) the binary relation B is defined by:

$$(m, n) \in B \iff m \in A_n$$

and,

c) B is effectively enumerable.

Show that there is an effectively enumerable subset C of $\mathbb{N}$ that is *not* effectively decidable.

We have been using the informal notions of *algorithm, effectively computable*, etc. It is surprising how much can be done informally (an example is Exercise 1.5.6). How can we make mathematically precise, these informal notions?

Question 1.5.26. Does there exist a mathematical characterization of the concepts of *effectively decidable* (for sets), and *effectively enumerable* (for sets), and *effectively computable* (for functions)?

Question 1.5.27. We have looked at two kinds of subsets of $\mathbb{N}$. The first consists of the class of subsets A of $\mathbb{N}$ that are effectively decidable. The second consists of the class of subsets of $\mathbb{N}$ that are effectively enumerable. Are these the same classes? (This is just a rephrasing of Question 1.5.6

Exercise 1.5.28. Let $f : A \xrightarrow{1-1} B$ be a 1-1 effectively computable function such that there is no algorithm for determining membership in the range of f. Let

$$D = \{n : \exists y > n\ [f(y) < f(n)]\}.$$

Show that:

a) D is effectively enumerable;

b) $\mathbb{N} \setminus D$ is infinite;

c) $\mathbb{N} \setminus D$ does not have an infinite subset that is effectively enumerable.

Note that we have not shown that there exists such a function f.

Chapter 2

Turing Machines

Alphabets

Let $A = \{a_1, \ldots, a_n\}$ be a finite set of distinct symbols.

a) A is called an *alphabet*.

b) A *word* (*string*) over A is a finite (possibly empty) sequence of symbols from A.

c) The empty word is denoted by ϵ.

d) If x and y are the words $b_1 \ldots b_k$ and $c_1 \ldots c_l$, respectively, then xy is the word $b_1 \ldots b_k c_1 \ldots c_l$, called the *concatenation* of x and y.

e) The *length* of the word x is the number of symbols in x and is denoted by $|x|$. So $|ab| = 2$ and $|\epsilon| = 0$.

f) The word aaa is abbreviated by a^3, etc.

Note that if x is a word then $\epsilon x = x\epsilon = x$.

Definition 2.0.1.

a) A^* is the set of all words over A.

b) A *language* over A is a subset of A^*, i.e. it is a set of words over A.

Example 2.0.2.

a) Let $A = \{a, b, c\}$. $abaa$ is a word over alphabet A.

b) $\{0,1\}^*$ is the set of all words over $A = \{0,1\}$. Some members of this set are: ϵ, 0, 1, 10, 00, 101, etc.

c) $A^* \setminus \{\epsilon\}$ is the set of all non-empty words over A.

2.1 Turing Machines: An Informal Definition

We define a class of objects called *Turing machines*. Each Turing machine may be regarded as an algorithm.[1]

The following is intended to give informal motivation for the precise definition of Turing machine that will be given later. Informally, a Turing machine consists of:

a) An *alphabet* $A = \{a_1, \ldots, a_n\}$, together with another symbol a_0 called the *blank* symbol.

b) A *tape* (on which calculations are performed) that is divided into squares and is infinite in both directions. The tape has a positive direction (to the right) and a negative direction (to the left). Each square of the tape is either blank (i.e. has a_0 printed on it) or has one of the symbols $a_1, \ldots, a_n$ printed on it. At any given time in a computation, all but a finite number of squares are blank.

c) A finite set $Q = \{q_0, \ldots\}$ of *states*. A state can be thought of as describing the internal configuration of the machine (in the old days this would be the arrangement of gears, gates, levers, etc.) at a particular instant.

d) A *read-write head* that at a given time is scanning (reading) the contents of one square of the tape, and is capable of replacing the symbol scanned by another symbol, or of moving to the left or to the right on the tape.

e) e) A finite ordered list of *instructions*.

Each instruction tells the computer what next state to enter and tells the read-write head to do one of

i) move one square to the left;

[1]Since Turing machines are a very special kind of abstract computer it is clear that many algorithms are not Turing machines. (The instructions comprising an algorithm might not be in the precise form of a Turing machine.)

ii) move one square to the right;

iii) replace the symbol scanned by another symbol (or leave it unchanged).

Let M be a Turing machine and w be a word over A. We describe very informally a *computation of* M *on input* w, as follows: Imagine the word w written on the tape and suppose all other squares on the tape are blank. The read-write head is scanning some square of the tape and M is in some initial state. By executing the instructions of M a finite number of times we go through a finite sequence of steps. At each step M is in a particular state, the head is scanning a particular square, and there is a specific word on the tape. (This word, which may have blanks in it, is surrounded by blanks.) This finite sequence of steps is called a *computation* if no further steps are possible (i.e. the procedure halts at the last step). Rather than spend more time making this precise we go to a formal treatment of these concepts.

2.2 Turing Machines: A Formal Definition

Definition 2.2.1. Fix in advance two symbols $\Rightarrow$ and $\Leftarrow$. These symbols will remain the same for all Turing machines. Let

$$\begin{aligned} &a_0\,, \\ &A = \{a_1, \ldots, a_n\}\,, \text{ and} \\ &Q = \{q_0, \ldots\} \end{aligned}$$

be such that the symbols $a_0, a_1, \ldots, a_n, q_0, \ldots, q_k, \Rightarrow, \Leftarrow$ are all distinct.
a_0 is called the *blank*. The symbols $a_0, a_1, \ldots, a_n$ are called *tape symbols*.[2]

1) An *instruction using states from* Q *and tape symbols from* $A \cup \{a_0\}$ is a quadruple:

$$q_i\; a_j\; b_k\; q_l$$

where

a) q_i and q_l are states (i.e. they belong to Q);

b) a_j is a tape symbol;

c) b_k is either a tape symbol, or the symbol $\Rightarrow$, or the symbol $\Leftarrow$.

[2]Informally, these are the symbols that can be written on the tape.

2) A *Turing machine* M *over alphabet* $A \cup \{a_0\}$ with blank a_0 and *states* Q is a finite ordered list of *instructions* using states from Q and alphabet A that satisfies the condition:
   for each state q and tape symbol a, there is at most one instruction that begins $q\ a\ ..\ ..$

3) If M is a Turing machine over alphabet $A \cup \{a_0\}$ with blank a_0 and states Q then the state that is the first component of the first instruction is called the *initial state* of M and is sometimes written q_{M}.

Remark 2.2.2. We may interpret an instruction informally as follows:

$$\begin{aligned} q\ a\ a'\ q' \quad & \text{:if in state } q \text{ scanning a square containing } a \text{ then} && (2.1)\\ & \text{replace } a \text{ by } a' \text{ and enter state } q'\ . && (2.2)\\ q\ a\ \Leftarrow\ q' \quad & \text{:if in state } q \text{ scanning a square containing } a \text{ then} && (2.3)\\ & \text{move the head one square to the left and enter state } q'\ . && (2.4)\\ q\ a\ \Rightarrow\ q' \quad & \text{:if in state } q \text{ scanning a square containing } a \text{ then} && (2.5)\\ & \text{move the head one square to the right and enter state } q'\ . && (2.6)\end{aligned}$$

Remark 2.2.3. In practice, if we rearrange the order of the instructions of a Turing machine we consider this to be the same machine, *provided* that the state that is the first component of the initial instruction remains the same. In other words, the order of the instructions serves only to single out the initial state.

Example 2.2.4. M consists of the instructions:

$$q_0\ 0\ 0\ q_1 \tag{2.7}$$
$$q_0\ 1\ 0\ q_0 \tag{2.8}$$
$$q_1\ 1\ \Rightarrow\ q_1 \tag{2.9}$$
$$q_2\ 0\ \Leftarrow\ q_3 \tag{2.10}$$
$$q_2\ 1\ \Rightarrow\ q_2 \tag{2.11}$$
$$q_3\ 0\ 0\ q_3 \tag{2.12}$$

The first instruction is $q_0\ 0\ 0\ q_1$. Its first component is q_0. So q_0 is the initial state. Note that there is no instruction beginning $q_1\ 0\ ..\ ..$ or $q_3\ 1\ ..\ ...$

Definition 2.2.5.

a) A *tape description* is a function $F\colon \mathbb{Z} \longrightarrow A \cup \{a_0\}$ such that $F(z) = a_0$ for all but finitely many z.

b) If F is a tape description and $z_0 \in \mathbb{Z}$ then $F^a_{z_0}$ is the tape description defined by:

$$F^a_{z_0} = \begin{cases} F(z), & \text{if } z \neq z_0; \\ a, & \text{if } z = z_0\ . \end{cases}$$

Remark 2.2.6. Think of each square of the tape as being numbered:

Then $F(z)$ is the symbol on the square whose number is z. Now suppose the symbol on square z_0 is changed to a. Then the new tape description would be $F^a_{z_0}$.

In our informal description of a Turing machine M, at a given stage:

a) M is in some state q,

b) there is some word written on the tape, and

c) the head is scanning a particular square.

The following definition makes this precise.

Definition 2.2.7. A *configuration* of M is a triple (F, q, z) such that

a) F is a tape description,

b) q is a state, and

c) $z \in \mathbb{Z}$.

A configuration (F, q, z) is a *terminal* configuration of M if there is no instruction of M that begins $q\ F(z)$ .. .. . Otherwise it is non-terminal.[3]

[3]Recall that there is at most one instruction that begins $q\ F(z)$ .. ... Consequently, if (F, q, z) is non-terminal then there is a *unique* instruction beginning $q\ F(z)$ .. ...

We next describe how the tape, state, and square scanned change as the instructions of M are executed.

Definition 2.2.8. The relation $\vdash_{\mathrm{M}}$ (or simply $\vdash$ for short) between configurations is defined as follows: $(F, q, z) \vdash (F', q', z')$ iff (F, q, z) is non-terminal, and if the line of the instruction of M beginning $q\ F(z)\ ..\ ..$ is:

$$q\ F(z)\ a\ p\ , \quad \text{then } F' = F_z^a \text{ and } z' = z \text{ and } p = q'\ ; \tag{2.13}$$

$$q\ F(z)\ \Rightarrow\ p\ , \quad \text{then } F' = F \text{ and } z' = z + 1 \text{ and } p = q'\ ; \tag{2.14}$$

$$q\ F(z)\ \Leftarrow\ p\ , \quad \text{then } F' = F \text{ and } z' = z - 1 \text{ and } p = q'\ . \tag{2.15}$$

Informally, if the configuration (F, q, z) describes the contents of the tape, the state, and the number of the square scanned at a particular time, and if $(F, q, z) \vdash (F', q', z')$ then (F', q', z') describes the contents of the tape, the state, and the square scanned after executing the unique applicable instruction of M.

Definition 2.2.9.

1) A *computation* of M is a finite sequence of configurations $C_1, \ldots, C_k$ such that:

   a) the second component of C_1 is the initial state q_0;

   b) $C_1 \vdash C_2, C_2 \vdash C_3, \ldots, C_{k-1} \vdash C_k$;

   c) C_k is terminal.

2) 2) If (F, q, z) is the terminal configuration of a computation, then q is called the *terminal state* of that computation.

Remark 2.2.10.

a) If (F, q, z) is the terminal configuration of a computation then there is no instruction beginning $q\ F(z)\ ..\ ...$

b) If $C_1, \ldots, C_k$ is a computation of M we say that the computation has *k steps*, or that it is a *computation of length k*. Each configuration is one step in the computation. We shall see that the same machine M can have different computations (for different inputs) with different lengths.

Remark 2.2.11. Formally, a Turing machine M is just a special kind of finite, ordered list of quadruples. However, by Remark 2.2.2, each quadruple of M can be interpreted informally as an instruction to carry out a certain effective procedure. So the set of quadruples comprising M defines an algorithm. A computation of M therefore corresponds to running this algorithm.

Examples of Turing Machines

It is sometimes convenient to use pictures as an aid in describing Turing machines and their action. The instruction: $q_i\ a\ b\ q_j$ can be described by the diagram

$q_i\ a\ \Rightarrow\ q_j$ can be described by the diagram

$q_i\ a\ \Leftarrow\ q_j$ can be described by the diagram.

The special case where $i = j$ gives, for example,

We shall not use this picture representation. Instead, we shall use the following way of describing a configuration. A configuration (F, q, z) can be described as follows:

$$z \tag{2.16}$$

$$q\ :\ \ldots a \ldots \tag{2.17}$$

$$\wedge \tag{2.18}$$

Here: the q on the left indicates thatMis in state q ; (2.19)
the position of $\wedge$ indicates that the head is scanning square (2.20)
number$z$, and$F(z) = a$; (2.21)
A (partial) picture of the tape description $F$is given by (2.22)

$$z \tag{2.23}$$

$$\ldots \quad a \ldots . \tag{2.24}$$

(Often we will leave the z, and sometimes the q, out of the picture.)

Exercise 2.2.12. Draw a picture of the configuration $(F, q_0, -1)$, where

$$F(0) = F(1) = F(2) = F(4) = F(5) = 1 \; ; \tag{2.25}$$

$$F(z) = 0 \quad \text{otherwise.} \tag{2.26}$$

Example 2.2.13 (The machine $\boldsymbol{T}_{\mathrm{R}}$)**.**

$$q_0\, a_0 \;\Rightarrow\; q_1 \tag{2.27}$$

$$q_0\, a_1 \;\Rightarrow\; q_1 \tag{2.28}$$

$$\vdots \tag{2.29}$$

$$q_o\, a_n \;\Rightarrow\; q_1 \tag{2.30}$$

This is a Turing machine with the two states q_0 and q_1, with q_0 the initial state, and tape symbols $a_0, \ldots, a_n$ (where a_0 is the blank). For each tape description F, we have $(F, q_0, z) \vdash (F, q_1, z + 1)$. (Since $(F, q_1, z + 1)$ is terminal there is no instruction beginning q_1 .. ..)

$$(F, q_0, z) \; , \; (F, q_1, z + 1)$$

is a computation. We can picture this computation as follows:

$$q_0 \; : \; \ldots ab \ldots \tag{2.31}$$

$$\wedge \tag{2.32}$$

$$q_1 \; : \; \ldots ab \ldots \tag{2.33}$$

$$\wedge \tag{2.34}$$

Thus $\boldsymbol{T}_{\mathrm{R}}$ moves the head one square to the right and halts (leaving the tape unchanged). In the special case where $A = \{1\}$ with 0 being the blank the instructions of $\boldsymbol{T}_{\mathrm{R}}$ are simply:

$$q_0\, 0 \;\Rightarrow\; q_1 \tag{2.35}$$

$$q_0\, 1 \;\Rightarrow\; q_1 \tag{2.36}$$

The following machines are written assuming that $A = \{1\}$, where 0 is the blank, but you should see for yourself in each case how to write the corresponding machine for an arbitrary alphabet A. Remember that with a tape description all but finitely many squares of the tape are blank (i.e. 0).

Example 2.2.14 (The machine $\boldsymbol{T}_{\mathrm{L}}$).

$$q_0\, 0 \Leftarrow q_1 \tag{2.37}$$
$$q_0\, 1 \Leftarrow q_1 \tag{2.38}$$

$\boldsymbol{T}_{\mathrm{L}}$ moves the head one square to the left and halts (leaving the tape unchanged).

Example 2.2.15 (The machine $\boldsymbol{T}_{\mathbf{0}}$).

$$q_0\ 1\ 0\ q_0$$

A (picture of a) computation of $\boldsymbol{T}_{\mathbf{0}}$ is:

$$q_0 \ : \ \ldots 0 \ldots \tag{2.39}$$
$$\wedge \tag{2.40}$$

Another computation is:

$$q_0 \ : \ \ldots 1 \ldots \tag{2.41}$$
$$\wedge \tag{2.42}$$
$$q_0 \ : \ \ldots 0 \ldots \tag{2.43}$$
$$\wedge \tag{2.44}$$

Thus $\boldsymbol{T}_{\mathbf{0}}$ writes 0 on the square scanned (if it isn't already 0) and halts. Note that the two computations above are computations of $\boldsymbol{T}_{\mathbf{0}}$ with different lengths.

Example 2.2.16 (The machine $\boldsymbol{T}_{\mathbf{1}}$).

$$q_0\ 0\ 1\ q_0$$

$\boldsymbol{T}_{\mathbf{1}}$ writes 1 on the square scanned (if it isn't already 1) and halts.

Example 2.2.17 (The machine $\boldsymbol{T}_{\mathrm{Lseek}\mathbf{0}}$).

$$q_0\, 0 \Leftarrow q_1 \tag{2.45}$$
$$q_0\, 1 \Leftarrow q_1 \tag{2.46}$$

$$q_1\, 1 \Leftarrow q_1$$

$\boldsymbol{T}_{\mathrm{Lseek}\mathbf{0}}$ finds the first 0 to the left of the starting position and halts.

Example 2.2.18 (The machine $\boldsymbol{T}_{\text{Rseek}\mathbf{0}}$)**.**

$$q_0\,0 \Rightarrow q_1 \tag{2.47}$$
$$q_0\,1 \Rightarrow q_1 \tag{2.48}$$

$$q_1\,1 \Rightarrow q_1$$

$\boldsymbol{T}_{\text{Rseek}\mathbf{0}}$ finds the first 0 to the right of the starting position and halts.

Example 2.2.19 (The machine $\boldsymbol{T}_{\text{Lseek}\mathbf{1}}$)**.**

$$q_0\,0 \Leftarrow q_1 \tag{2.49}$$
$$q_0\,1 \Leftarrow q_1 \tag{2.50}$$

$$q_1\,0 \Leftarrow q_1$$

$\boldsymbol{T}_{\text{Lseek}\mathbf{1}}$ finds the first 1 to the left of the starting position (if there is one) and halts; if there is none the head keeps moving to the left and no computation exists.

Example 2.2.20 (The machine $\boldsymbol{T}_{\text{Rseek}\mathbf{1}}$)**.**

$$q_0\,0 \Rightarrow q_1 \tag{2.51}$$
$$q_0\,1 \Rightarrow q_1 \tag{2.52}$$

$$q_1\,0 \Rightarrow q_1$$

$\boldsymbol{T}_{\text{Rseek}\mathbf{1}}$ behaves just like $\boldsymbol{T}_{\text{Lseek}\mathbf{1}}$, except that it searches to the right.

2.3 Submachines

Let M be a Turing machine over some alphabet $A \cup \{a_0\}$ with states $Q = \{q_0, \ldots, q_m\}$. Let $P = \{p_o, \ldots, p_m\}$ and M′ be the machine with alphabet $A \cup \{a_0\}$ with states P that is obtained from M by replacing (for each i) each occurrence of q_i in an instruction by p_i. Then M′ behaves on a given initial tape description just like M, so we can always "relabel" our states without affecting the operation of the machine. In particular, if M_1 and $\boldsymbol{M}_2$ are Turing machines with states P and Q respectively, we can assume without loss of generality that $P \cap Q = \emptyset$. This is useful when we want to combine several machines into a single machine.

Definition 2.3.1. Let M_1 and M_2 be Turing machines over the same alphabet. $\rightarrow \mathrm{M}_1 \rightarrow \mathrm{M}_2$ is the Turing machine obtained by performing the following steps:

a) Relabel the states of M_2 so that M_1 and (the new) M_2 have no states in common;

b) Write down the instructions for M_1, followed by the instructions for M_2;

c) For each pair $q\ a$ (where q is a state of M_1), *such that there is no instruction of M_1 that begins $q\ a$ .. ..*, add (at the end of the list of instructions) the instruction $q\ a\ a\ q_{\mathrm{M}_2}$, where q_{M_2} is the initial instruction of M_2.

Remark 2.3.2.

1. On a given initial configuration, the machine $\rightarrow \boldsymbol{M}_1 \rightarrow \mathrm{M}_2$ acts just like M_1 until $\boldsymbol{M}_1$ halts; then it acts like M_2 with the terminal tape and head position of M_1 the same as the initial tape and head position of M_2.

2. Because of the way in which submachines have been defined (see Definition 2.3.1) the number of steps in the computation is obtained by adding the number of steps in the computation of $\mathbf{M}_1$ and M_2.

Remark 2.3.3. $\rightarrow \mathrm{M}_1 \rightarrow \mathrm{M}_2$ is not uniquely determined since we have not specified how to choose the states of $\boldsymbol{M}_2$. But it doesn't matter how we choose these states as long as they are disjoint from those of M_1.

Exercise 2.3.4. Write out explicitly the instructions for the machine $\rightarrow \boldsymbol{T}_{\mathrm{Lseek}\mathbf{0}} \rightarrow \boldsymbol{T}_{\mathrm{R}}$.

Definition 2.3.5. Let M_1, M_2, and M_3 be Turing machines on alphabet $A \cup \{a_0\}$, and let $a, b \in A \cup \{a_0\}$. The Turing machine

$$\begin{tikzcd} \rightarrow\mathbf{M}_1 \arrow[r, "\text{if } a"] \arrow[d, "\text{if } b"] & \mathrm{M}_2 \\ \mathrm{M}_3 \end{tikzcd} \qquad (2.53)\ (2.54)\ (2.55)$$

is obtained as follows:

a) Relabel the states of the machines M_2 and M_3 so that the three machines have no states in common;

b) Write down the instructions for M_1, followed by those of (the new) M_2 and (the new) M_3;

c) For each state q of M_1 *such that there is no instruction of M_1 that begins* $q\ a$ .. .. add, at the end of the list of instructions, the instruction $q\ a\ a\ q_{M_2}$;

d) For each state q of M_1 *such that there is no instruction of M_1 that begins* $q\ b$ .. .. add, at the end of the list of instructions, the instruction $q\ b\ b\ q_{M_3}$.

On a given initial configuration, the above machine behaves just like $\boldsymbol{M}_1$ until M_1 halts. Then, if M_1 halts in a configuration with head scanning a square containing a, the machine behaves like M_2 with the terminal tape and head position of M_1 the same as the initial tape and head position of $\boldsymbol{M}_2$. Similarly, if M_1 halts in a configuration with head scanning a square containing b, the machine behaves like M_3 with the terminal tape and head position of M_1 the same as the initial tape and head position of M_3. And if M_1 halts in a configuration with head scanning a square not containing a or b, then the new machine halts.

Other variations of composite machines are possible, as described below. It is now possible to link together a number of Turing machines to form more complicated Turing machines. You should make sure that if you had to you could write out the instructions of a single machine that corresponds to a given diagram.

Example 2.3.6. 6.4 (The machine $\boldsymbol{T}_{\text{Lend}}$). It moves left until it finds at least two consecutive 0's, (i.e. ...00...) and then stops at the rightmost one, (i.e. it stops at ...00...

$$\text{if } 1 \tag{2.56}$$

$$\rightarrow \boldsymbol{T}_{\text{Lseek0}} \longrightarrow \boldsymbol{T}_{\text{L}} \tag{2.57}$$

$$\downarrow \text{if } 0 \tag{2.58}$$

$$\boldsymbol{T}_{\text{R}} \tag{2.59}$$

$$\tag{2.60}$$

Exercise 2.3.7. Write out the list of instructions for the machine in Exercise ???

Example 2.3.8 (The machine $\boldsymbol{T}_{\text{Rend}}$)**.** It moves right until it finds $\ldots 00 \ldots$ and then stops at $\ldots 00 \ldots$

$$\begin{tikzcd} \arrow[r] & \boldsymbol{T}_{\text{Rseek0}} \arrow[r] & \boldsymbol{T}_{\text{R}} \arrow[l, bend right, "\text{if } 1"'] \arrow[d, "\text{if } 0"] \\ & & \boldsymbol{T}_{\text{L}} \end{tikzcd} \tag{2.61–2.65}$$

Example 2.3.9 (The machine $\boldsymbol{T}_{\text{seek1}}$)**.** It finds a 1 on the tape and halts, staring at it. If the tape has no 1's it never halts. The terminal tape description is meant to be the same as the initial tape description.

Note: The head must search both left and right; but how does it know when to turn around and look in the other direction? It leaves a *marker* that will later be erased. The only usable marker available is 1.

$$\begin{tikzcd} \arrow[r] & \boldsymbol{T}_L \arrow[r, "\text{if } 0"] \arrow[d, "\text{if } 1"] & \boldsymbol{T}_1 \arrow[r] & \boldsymbol{T}_R \arrow[r, "\text{if } 0"] & \boldsymbol{T}_1 \arrow[r] & \boldsymbol{L}_{\text{Lseek1}} \arrow[r] & \boldsymbol{T}_0 \arrow[r] & \boldsymbol{T}_L \arrow[r, "\text{if } 0"] \arrow[d, "\text{if } 1"] & \boldsymbol{T}_1 \arrow[r] & \boldsymbol{T}_{\text{Rseek1}} \arrow[r] & \boldsymbol{T}_0 \\ & \boldsymbol{T}_{\text{Lseek1}} \arrow[d] & & & & & & \boldsymbol{T}_{\text{Rseek1}} \arrow[d] \\ & \boldsymbol{T}_0 \arrow[d] & & & & & & \boldsymbol{T}_0 \arrow[d] \\ & \boldsymbol{T}_{\text{Rseek1}} & & & & & & \boldsymbol{T}_{\text{Lseek1}} \end{tikzcd}$$

Example 2.3.10 (The machine $\boldsymbol{T}_{\text{Ltrans}}$)**.** This is a machine that, on an initial configuration of the form

$$\ldots a01^{b+1}0 \ldots \tag{2.66}$$
$$\wedge \tag{2.67}$$

has terminal configuration of the form

$$\ldots a1^{b+1}00 \ldots \tag{2.68}$$
$$\wedge \tag{2.69}$$
$$\tag{2.70}$$

$$\rightarrow \boldsymbol{T}_{\text{R}} \longrightarrow \boldsymbol{T}_{\mathbf{1}} \longrightarrow \boldsymbol{T}_{\text{Rseek}\mathbf{0}} \longrightarrow \boldsymbol{T}_{\text{L}} \longrightarrow \boldsymbol{T}_{\mathbf{0}}$$

Remark 2.3.11.

a) In this and (most) future examples, the infinite segments of the tape indicated by . . . are left unchanged during the computation.

b) We don't care what the machine does if the initial configuration is not of the above form.

Example 2.3.12 (The machine $\boldsymbol{T}_{\mathrm{Lshift}}$)**.** This machine on initial configuration

$$\begin{array}{c} \ldots 01^{x+1}01^{y+1}0\ldots \quad (2.71)\\ \quad\quad\quad\quad\wedge \quad\quad\quad (2.72)\end{array}$$

has terminal configuration

$$\begin{array}{c} \ldots 01^{y+1}000^{x+1}\ldots \quad (2.73)\\ \quad\quad\wedge \quad\quad\quad\quad\quad (2.74)\end{array}$$

$$\begin{tikzcd} \arrow[r] & \boldsymbol{T}_{\mathrm{Lseek0}} \arrow[r] & \boldsymbol{T}_{\mathrm{L}} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \boldsymbol{T}_{\boldsymbol{0}} \arrow[r] & \boldsymbol{T}_{\mathrm{Ltrans}} \\ & & \boldsymbol{T}_{\mathrm{Ltrans}} & & \end{tikzcd} \quad (2.75)\text{–}(2.77)$$

Example 2.3.13 (The machine $\boldsymbol{T}_{\mathrm{fin}}$)**.** This machine on initial configuration

$$\begin{array}{c} \ldots 001^{x_0+1}01^{x_1+1}0\ldots 01^{x_{n-1}+1}01^{y+1}0\ldots \quad (2.78)\\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\wedge \quad\quad\quad (2.79)\end{array}$$

has terminal configuration

$$\begin{array}{c} \ldots 1^{y+1}00^{2+x_0+1+x_1+1+\cdots+x_{n-1}+1+n}\ldots \quad (2.80)\\ \wedge \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (2.81)\end{array}$$

$$\rightarrow \boldsymbol{T}_{\text{Lseek}\mathbf{0}} \longrightarrow \boldsymbol{T}_{\text{L}} \xrightarrow{\text{if } 1} \boldsymbol{T}_{\text{R}} \longrightarrow \boldsymbol{T}_{\text{Rseek}\mathbf{0}} \longrightarrow \boldsymbol{T}_{\text{Lshift}} \tag{2.82}$$

$$\downarrow \text{ if } 0 \tag{2.83}$$

$$\boldsymbol{T}_{\text{Ltrans}} \tag{2.84}$$

$$\downarrow \tag{2.85}$$

$$\boldsymbol{T}_{\text{Lseek}\mathbf{0}} \tag{2.86}$$

$$\downarrow \tag{2.87}$$

$$\boldsymbol{T}_{\text{L}} \tag{2.88}$$

$$\downarrow \tag{2.89}$$

$$\boldsymbol{T}_{\text{Ltrans}} \tag{2.90}$$

Example 2.3.14 (The machine $\boldsymbol{T}_{\text{copy}}$)**.** This machine on initial configuration

$$\ldots 01^{x+1}00^{x+2}\ldots \tag{2.91}$$

$$\wedge \tag{2.92}$$

has terminal configuration

$$\ldots 1^{x+1}01^{x+1}0\ldots \tag{2.93}$$

$$\wedge \tag{2.94}$$

Note: The machine must keep track of how many 1's have been copied at a particular time. This is done by temporarily replacing a 1 by a 0 and then later replacing the 0 by a 1.

$$\begin{tikzcd}
{} \arrow[r] & \boldsymbol{T}_{\text{Lseek0}} \arrow[r] & \boldsymbol{T}_{\text{R}} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \boldsymbol{T}_{\mathbf{0}} \arrow[r] & \boldsymbol{T}_{\text{Rseek0}} \arrow[r] & \boldsymbol{T}_{\text{Rseek0}} \arrow[r] & \boldsymbol{T}_{1} \arrow[r] & \boldsymbol{T}_{\text{Lseek0}} \arrow[r] & \boldsymbol{T}_{\text{Lseek0}} \arrow[r] & \boldsymbol{T}_{\mathbf{1}} \\
 & & \boldsymbol{T}_{\text{Rseek0}} & & & & & & &
\end{tikzcd}$$

Definition 2.3.15. For k a positive integer M^k is an abbreviation of

$$\rightarrow \underbrace{\mathrm{M} \rightarrow \boldsymbol{M} \rightarrow \cdots \rightarrow \mathrm{M}}_{k \text{ times}}$$

Example 2.3.16 (The machine $\boldsymbol{T}_{\boldsymbol{n}\text{copy}}$)**.** This is a different machine for each positive integer n. This machine on initial configuration

$$\ldots 01^{x_0+1}01^{x_1+1}0\ldots 01^{x_{n-1}+1}00^{x_0+2}\ldots \tag{2.95}$$

$$\wedge \tag{2.96}$$

has terminal configuration

$$\ldots 01^{x_0+1}01^{x_1+1}0\ldots 01^{x_{n-1}+1}01^{x_0+1}0\ldots \tag{2.97}$$
$$\wedge \tag{2.98}$$

$$\begin{tikzcd}
\arrow[r] & \boldsymbol{T}^{\boldsymbol{n}}_{\mathrm{Lseek0}} \arrow[r] & \boldsymbol{T}_{\mathrm{R}} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \boldsymbol{T_0} \arrow[r] & \boldsymbol{T}^{\boldsymbol{n+1}}_{\mathrm{Rseek}\mathbf{0}} \arrow[r] & \boldsymbol{T_1} \arrow[r] & \boldsymbol{T}^{\boldsymbol{n+1}}_{\mathrm{Lseek}\mathbf{0}} \arrow[r] & \boldsymbol{T_1} \\
& & \boldsymbol{T}^{\boldsymbol{n}}_{\mathrm{Rseek0}} & & & & &
\end{tikzcd} \tag{2.99–2.101}$$

Exercise 2.3.17. Define a Turing machine M that if started on a completely blank tape (0 is the blank) will print *in both directions* the infinite sequence

$$\ldots 110011001100\ldots$$

M never halts of course.

Exercise 2.3.18. Define a Turing machine that if started on a completely blank tape (0 is the blank) will print to the right the infinite sequence

$$\ldots 10110111011110\ldots$$

Exercise 2.3.19. Show that there does not exist a Turing machine that if started at an arbitrary position on a tape that has at least one 1 on it, will stop at the leftmost 1. (Note that a tape has at most finitely many non-blank squares.)

Exercise 2.3.20. Let M be a Turing machine over alphabet $\{0, 1\}$ with 0 the blank. $P(\mathrm{M})$, the *productivity* of M, is the number of 1's on the tape if M halts when started on a completely blank tape. The productivity of M is 0 if M does not halt when started on a completely blank tape. Let

$$p(n) = \max\{P(\mathrm{M}) : \text{M is a Turing machine with } n \text{ states}\}\ .$$

For $n = 1, 2, 3, 4, 5$ *try* to calculate $p(n)$ by finding, for each n, a Turing machine M with n states such that $P(\mathrm{M})$ is as large as possible.

2.4 Decidability and Acceptability

Definition 2.4.1. Let F be a tape description. The word $w = b_0 \dots b_k$ *lies on F beginning at m and ending at n* (where $m, n \in \mathbb{Z}$) if

$$F(m) = b_0, F(m+1) = b_1, \dots, F(m+k) = b_k \text{ , where } n = m+k \text{ .}$$

The corresponding picture is:

$$m \qquad n \tag{2.102}$$

$$\dots b_0 b_1 \dots b_k \dots \tag{2.103}$$

Definition 2.4.2. Let M be a Turing machine on alphabet $A \cup \{0\}$, where 0 is the blank, and w be a non-empty word over A.

a) M *accepts w* if whenever:

   i) F is a tape description,

   ii) $m, n \in \mathbb{Z}$,

   iii) w lies on F beginning at m and ending at n, and

   iv) $F(i) = 0$ whenever $i < m$ or $i > n$,

   then there is a computation of M beginning with the initial configuration $(F, q_{\mathrm{M}}, n+1)$. We say then that M *halts on input w*.

b) The *set accepted by* M is the set

$$\{w \in A^* \setminus \{\epsilon\} : w \text{ is accepted byM}\} \text{ .}$$

That is, the set accepted by M is the set of all non-empty words w over A such that M halts on input w.

Remark 2.4.3.

a) In terms of a picture, M accepts w iff w is a non-empty word over A and M halts when started on the initial configuration:

$$q_{\mathrm{M}} \ : \ {}^{\infty}0w00^{\infty} \text{ .} \tag{2.104}$$

$$\wedge \tag{2.105}$$

   From now on we will use this picture representation. [4]

[4]The expression ${}^{\infty}01$ means that there are all 0's to the left of the 1, and 10^{∞} means that there are all 0's to the right of the 1.

b) If M accepts the set X of non-empty words then for all non-empty words w,

$$w \in X \implies \text{M halts on input } w \text{ and} \tag{2.106}$$
$$w \notin X \implies \text{M does not halt on input } w\ . \tag{2.107}$$

Definition 2.4.4. Let M be a Turing machine with two special states q_Y and q_N. Let X be a set of non-empty words over A.

a) M *decides membership in* X if for every non-empty word w over A and initial configuration:

$$q_{\mathrm{M}} \ :\ \ {}^{\infty}0w\underset{\wedge}{0}0^{\infty}$$

there is a computation beginning with the above configuration that has terminal state:

$$\begin{aligned} q_Y &\text{ if } w \in X\ , \\ q_N &\text{ if } w \notin X\ . \end{aligned}$$

b) X is *decidable* if there is some Turing machine that decides membership in X.

Remark 2.4.5.

a) $A^* \smallsetminus \{\epsilon\}$ is decidable.

b) $\emptyset$ is decidable.

Let $X \subseteq A^* \smallsetminus \{\epsilon\}$.

c) X is decidable iff $(A^* \smallsetminus \{\epsilon\}) \smallsetminus X$ is decidable.

d) If X is decidable then there exists a Turing machine that accepts X.

Remark 2.4.6 (Informal)**.** If X is decidable then X is effectively decidable (i.e. there is an algorithm for determining membership in X).

Question 2.4.7. Suppose $X \subseteq A^* \smallsetminus \{\epsilon\}$, and suppose that X is effectively decidable. Must X be decidable?

Exercise 2.4.8. Let M be a Turing machine over $A = \{1\}$, where 0 is the blank, and let X be the set accepted by M. Explain why X is effectively enumerable by describing an algorithm that lists the members of X.

Question 2.4.9. Suppose that X is a set of non-empty words over $A = \{1\}$ that is effectively enumerable. Must there be a Turing machine that accepts X?

Question 2.4.10. Suppose that X is accepted by some Turing machine. Is X necessarily decidable?

Exercise 2.4.11. Let $A = \{1, 2\}$ and $X = \{1^n 2^n : n \geq 1\}$. Show that X is decidable.

Exercise 2.4.12. Let $A = \{1, 2\}$ and X be the set of non-empty words over A that have exactly twice as many 1's as 2's. Show that X is decidable.

2.5 Complexity of Computation

Suppose that X is a decidable set of non-empty words over A. There are many Turing machines that decide membership in X. Some of these machines may run more 'efficiently' than others. How do we measure efficiency? One natural measure is in terms of the number of steps in the computation (which changes with changes in the input). Another measure is in terms of the amount of tape used in the computation. This leads to the following definitions.

Definition 2.5.1. Let M be a Turing machine with alphabet $A \cup \{0\}$.

a) Let $f \colon \mathbb{N} \to \mathbb{N}$. M *runs in time bounded by the function* f if for every non-empty word w over A, there is a computation $C_1, \ldots, C_k$, where C_1 is the initial configuration

$$^{\infty}0w00^{\infty} \tag{2.108}$$

$$\wedge \tag{2.109}$$

such that $k \leq f(|w|)$.

b) M *runs in polynomial time* if there is some polynomial p such that M runs in time bounded by p.

In a computation $C_1, \ldots, C_k$ the head can move at most k squares from its initial position. The number of squares visited by the head during a computation is called the *space used by the computation*. (If the same square is visited more than once, it is only counted once.)

Definition 2.5.2. Let M be a Turing machine on alphabet $A \cup \{0\}$.

a) Let $f\colon \mathbb{N} \to \mathbb{N}$. M *runs in space bounded by f* if for every non-empty word w over A, there is a computation beginning with the initial configuration

$$^{\infty}0w00^{\infty} \tag{2.110}$$

$$\wedge \tag{2.111}$$

that uses space bounded by $f(|w|)$ (i.e. the number of squares visited is $\leq f(|w|)$).

b) M *runs in polynomial space* if for some polynomial p, M runs in space bounded by p.

Definition 2.5.3. Let X be a set of non-empty words over A.

a) X is *decidable in polynomial time* if there exists a Turing machine that runs in polynomial time that decides membership in X.

b) X is *decidable in polynomial space* if there is a Turing machine that runs in polynomial space that decides membership in X.

Remark 2.5.4. It is clear that if X is decidable in polynomial time then X is decidable in polynomial space.

Open Problem If X is decidable in polynomial space is X necessarily decidable in polynomial time?

Remark 2.5.5. Suppose that $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ are Turing machines. If on initial configuration C_1, $\boldsymbol{M}_1$ halts in exactly n_1 steps with terminal configuration C_2, and on initial configuration C_2, $\boldsymbol{M}_2$ halts in exactly n_2 steps, then the machine $\to \boldsymbol{M_1} \to \boldsymbol{M_2}$ on initial configuration C_1 halts in exactly $n_1 + n_2$ steps. For example, the machine $\boldsymbol{T}_R \to \boldsymbol{T}_L$ always halts in exactly 4 steps. You will want to use this information in the following exercise.

Exercise 2.5.6. Find the function p such that the computation of $\boldsymbol{T}_{\text{copy}}$ on initial configuration

$$^{\infty}01^{n}00^{\infty} \tag{2.112}$$
$$\wedge \tag{2.113}$$

has exactly $p(n)$ steps.

Example 2.5.7. Let M be a Turing machine that on initial configuration

$$01^{n+1}00^{\infty} \tag{2.114}$$
$$\wedge \tag{2.115}$$

has terminal configuration

$$01^{n+1}01^{2^{n+1}}00^{\infty} \tag{2.116}$$
$$\wedge \tag{2.117}$$

Then M does not run in polynomial space and hence not in polynomial time. (We have not yet seen how to define such an M, but we will!)

Exercise 2.5.8. Suppose that M is a Turing machine on A that runs in *space* bounded by f. Show that there are $m, n \in \mathbb{N}$ such that M runs in *time* bounded by g, where $g(x) = m \cdot n^{f(x)}$.

Hint: Things to consider are the number of members in A, and the number of states of M.

Exercise 2.5.9. Let $A = \{1\}$.

a) Find a Turing machine M that on initial configuration

$$^{\infty}01^{n+1}00^{\infty} \tag{2.118}$$
$$\wedge \tag{2.119}$$

has terminal configuration

$$^{\infty}01^{n+1}01^{2n+1}00^{\infty} \tag{2.120}$$
$$\wedge\ . \tag{2.121}$$

Find the simplest polynomial p such that M runs in time bounded by p.

b) Find a Turing machine M that on initial configuration

$$^{\infty}01^{n+1}00^{\infty} \tag{2.122}$$
$$\wedge \tag{2.123}$$

has terminal configuration

$$^{\infty}01^{n+1}01^{n^2+1}00^{\infty} \tag{2.124}$$
$$\wedge\ . \tag{2.125}$$

Again, find the simplest polynomial p.

c) Find a Turing machine M that on initial configuration

$$^{\infty}01^{n+1}00^{\infty} \tag{2.126}$$
$$\wedge \tag{2.127}$$

has terminal configuration

$$^{\infty}01^{n+1}01^{2^n+1}00^{\infty} \tag{2.128}$$
$$\wedge\ . \tag{2.129}$$

Again, find the simplest polynomial p.

2.6 Turing Computable Functions

Suppose f is a partial function of one variable. We would like to make precise what it means for the Turing machine M to compute f. There is a difficulty, however. The arguments of f are from the infinite set $\mathbb{N}$. But a Turing machine has only a finite alphabet. This problem is easily overcome by representing integers to some base. Any base will do, but the easiest is to use base 1 (or something close to it): we represent the integer n by the word 1^{n+1}. So 0 is represented by 1. 1 is represented by 11, etc. The k-tuple $(x_0, \ldots, x_{k-1})$ of integers is represented by the word

$$1^{x_0+1}01^{x_1+1}0\ldots1^{x_{k-1}+1}\ .$$

Definition 2.6.1. Let f be a k-ary partial function.

1) The Turing machine M *computes* f if for every k-tuple $(x_0, \ldots, x_{k-1})$,

a) if $f(x_0, \ldots, x_{k-1})$ is defined then there is a computation with initial configuration

$$\underbrace{\qquad\qquad\qquad\ldots}_{\text{not necessarily all 0's}} 01^{x_0+1}0\ldots01^{x_{k-1}+1}00^{\infty}$$
$$\wedge$$

and terminal configuration

$$\ldots01^{x_0+1}0\ldots01^{x_{k-1}+1}01^{f(x_0,\ldots,x_{k-1})+1}00^{\infty}$$
$$\wedge \;.$$

(The tape to the left of the leftmost 0 in the picture of the terminal configuration is identical to that in the initial configuration.)

b) If $f(x_0, \ldots, x_{k-1})$ is not defined then there does not exist a computation beginning with the initial configuration in (a).

2) f is a *partial Turing computable* function if there is some Turing machine that computes f.

3) f is *Turing computable* if f is partial Turing computable and f is total.

Proposition 2.6.2. *Let s be the successor function: $s(x) = x + 1$. Then s is Turing computable.*

Proof. The following machine computes s.

$$\to \mathrm{T}_{\mathrm{copy}} \longrightarrow \mathrm{T}_1 \longrightarrow \mathrm{T}_{\mathrm{R}} \;.$$

□

Definition 2.6.3. For each $k > 0$ and $n \geq 0$, Cs_n^k is the k-ary function:

$$\mathrm{Cs}_n^k(x_0, \ldots, x_{k-1}) = n \;.$$

Exercise 2.6.4. Show that Cs_n^k is Turing computable.

Definition 2.6.5. For each $n > 0$ and $i < n$, Pr_i^n is the n-ary function:

$$\mathrm{Pr}_i^n(x_0, \ldots, x_{n-1}) = x_i \;.$$

In particular, Pr_0^1 is the identity function.

Proposition 2.6.6. *Pr^n_i is Turing computable.*

Proof. $\boldsymbol{T}_{(n-i)\text{ copy}}$ computes Pr^n_i. □

Definition 2.6.7. Let R be a k-ary relation on $\mathbb{N}$. C_R, the *characteristic function of R*, is defined by:

$$\mathrm{C}_R(x_0, \ldots, x_{k-1}) = \begin{cases} 1, & \text{if } (x_0, \ldots, x_{k-1}) \in R; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 2.6.8.

a) The relation R is *Turing computable* if its characteristic function is Turing computable.

b) The relation R is *Turing semi-computable* if it is equal to the domain of some partial Turing computable function.

Remark 2.6.9. From the association of algorithms with Turing machines it is clear that:

a) If f is a partial Turing computable function then f is a partial effectively computable function.

b) If the relation R is Turing computable then R is effectively decidable.

c) If the relation R is Turing semi-computable, then R is effectively enumerable. (Why?!)

Question 2.6.10.

a) Is the converse of 2.6.9 (a) true?

b) Is the converse of 2.6.9(b) true?

What methods could one use to show this?

It is fairly clear that we can design Turing machines to compute a wide variety of functions. In some cases these machines are apt to be rather complicated. Instead of showing, one at a time, that a lot of functions are Turing computable, we seek a mathematical characterization of the class of Turing machines in terms of partial recursive functions. This will be done in the next chapter.

Exercise 2.6.11. Define Turing machines that compute the following functions:

a) $+$

b) f, where for all n, $f(n) = 3n$

c) $\times$

Exercise 2.6.12. This exercise is a continuation of Exercise 2.3.20. It gives a concrete example of a total function that is not Turing computable.

Let $n > 0$. Show:

a) There is a Turing machine $\boldsymbol{P_n}$ with $n + 1$-states that, when started on a completely blank tape, has terminal configuration

$$^{\infty}01^{n}00^{\infty}\ . \tag{2.130}$$

$$\wedge \tag{2.131}$$

b) $p(n) \geq n$.

c) $p(n+2) > p(n)$.

d) $p(n+18) \geq 2n$. (*Hint:* Look at $\mathrm{T_{copy}}$.)

Suppose that p is Turing computable by a Turing machine $\boldsymbol{T_p}$ with k states. Show:

e) $p(n+2+2k) \geq p(p(n))$. (*Hint:* Look at $\rightarrow \boldsymbol{P_{n+1}} \longrightarrow \boldsymbol{T_p} \longrightarrow \boldsymbol{T_p}$.)

f) $p(n+20+2k) \geq p(2n)$.

g) Conclude that p is not Turing computable.

2.7 Computations in Base 2

The modification of base 1 used in section ?? works for representing number-theoretic functions, but it is not very efficient. The representation of the integer n uses $n+1$ squares of the tape. In the following exercises let $A = \{0, 1\}$ and b be the blank symbol. We can think of a non-empty word w on A as being the name, written in base 2, of a non-negative integer n_w. (We disregard 0's at the beginning, so for example, 101 and 0101 are both names for 5.) There is a representation of a positive integer n with not more than $[\log_2(n)] + 1$ symbols, where $[r]$ is the integer part of the real number r. In third(?) grade you learned algorithms for addition, subtraction, multiplication, and division, in base 10. These algorithms are essentially the same, but slightly simpler, in base 2. The following exercises will help you decide whether or not you should repeat third grade. Each of the machines you construct below should run in polynomial time with the possible exception of the last one. For each of your machines, indicate the lowest degree of the polynomial that works for your machine. In these exercises, v and w are non-empty words on $A = \{0, 1\}$.

Exercise 2.7.1. Define a Turing machine $\boldsymbol{T}_{\mathbf{>0}}$ that does the following: $\boldsymbol{T}_{>0}$ has two special states $q_{\boldsymbol{Y}}$ and $q_{\boldsymbol{N}}$, and on initial configuration

$$\ldots bwb \ldots \tag{2.132}$$
$$\qquad\wedge \tag{2.133}$$

has terminal configuration

$$q_{\boldsymbol{Y}} \ : \ \ldots bwb \ldots \tag{2.134}$$
$$\qquad\qquad\wedge \tag{2.135}$$

if $n_w > 0$, and has terminal configuration

$$q_{\boldsymbol{N}} \ : \ \ldots bwb \ldots \tag{2.136}$$
$$\qquad\qquad\wedge \tag{2.137}$$

if $n_w = 0$.

Exercise 2.7.2. Define a Turing machine $\boldsymbol{T}_<$ that does the following: $\boldsymbol{T}_<$ has two special states $q_{\boldsymbol{Y}}$ and $q_{\boldsymbol{N}}$, and on initial configuration

$$\ldots bvbwb \ldots \tag{2.138}$$
$$\qquad\wedge \tag{2.139}$$

has terminal configuration

$$q_{\boldsymbol{Y}} \ : \ \ldots bvbwb \ldots \tag{2.140}$$
$$\qquad\qquad\qquad\wedge \tag{2.141}$$

if $n_w \leq n_v$, and has terminal configuration

$$q_N \ : \ \dots bvbwb \dots \tag{2.142}$$
$$\wedge \tag{2.143}$$

otherwise.

Exercise 2.7.3. Define Turing machines $\boldsymbol{T}_{\text{sub1}}$ and $\boldsymbol{T}_{\text{add1}}$ such that: $\boldsymbol{T}_{\textbf{sub1}}$ on initial configuration

$$\dots bvb \dots \tag{2.144}$$
$$\wedge \tag{2.145}$$

where $n_v > 0$, has terminal configuration

$$\dots bv'b \dots \tag{2.146}$$
$$\wedge \tag{2.147}$$

where $n_{v'} = n_v - 1$ (i.e. $\boldsymbol{T}_{\text{sub1}}$ subtracts 1 from v (working in base 2).[5] $\boldsymbol{T}_{\textbf{add1}}$ behaves like $\boldsymbol{T}_{\textbf{sub1}}$, except that it adds 1 instead of subtracting 1.

Exercise 2.7.4 (Subtraction in Base 2)**.** Define a Turing machine $\boldsymbol{T}_{\text{sub}}$ over A that, on initial configuration

$$\dots bvbwb \dots \ , \tag{2.148}$$
$$\wedge \tag{2.149}$$

where $n_w \leq n_v$, has terminal configuration

$$\dots bv'bwb \dots \tag{2.150}$$
$$\wedge \tag{2.151}$$

where $n_{v'} = n_v - n_w$.

Exercise 2.7.5. Define a Turing machine $\boldsymbol{T}_{\text{rem}}$ that, on initial configuration

$$\dots bvbwb \dots \tag{2.152}$$
$$\wedge \tag{2.153}$$

[5]For example if the initial tape description is $\dots b100b \dots$ then the terminal tape description will be $\dots b011b \dots$.

where $n_w \le n_v$, has terminal configuration

$$\ldots bv'bwb \ldots\ , \tag{2.154}$$
$$\wedge \tag{2.155}$$

where $n_{v'}$ = the remainder when n_v is divided by n_w.

Exercise 2.7.6. Define a Turing machine $\boldsymbol{T}_{\text{factor}}$ that, on initial configuration

$$\ldots bvbwb \ldots \tag{2.156}$$
$$\wedge \tag{2.157}$$

where $n_w \le n_v$, has terminal state

q_Y if n_w is a factor of n_v;

q_N otherwise.

Exercise 2.7.7. Define a Turing machine that decides membership in the set of primes (when they are expressed in base 2). Can you arrange it so that your machine runs in polynomial time? [6]

Exercise 2.7.8. Define a Turing machine that, when started on a completely blank tape, runs through in order (of magnitude in base 2) *all* tape descriptions of the form ${}^\infty bwbb^\infty$, where w is a word over $A = \{0, 1\}$ that (except for 0 itself) do not begin with 0). This machine never halts of course. So it lists in order, 0b1b10b11b100b101b110b111, 1000, etc.

2.8 Primitive Recursive Functions

We use $\bar{x}$ to stand for a tuple. For example, if $\bar{x} = x_1, x_2, x_3$, then $f(\bar{x})$ means $f(x_1, x_2, x_3)$. Often $\bar{x}$ will be an n-tuple, $\bar{x} = x_1, \ldots, x_n$, a k-tuple, $\bar{x} = x_1, \ldots, x_k$, etc. Recall that a partial n-ary *number-theoretic* function is a function whose domain is a subset of $\mathbb{N}^n$ and whose range is a subset of $\mathbb{N}$. It is *total* if its domain is all of $\mathbb{N}^n$. Also recall that $f(\bar{x}) \downarrow$ means that the tuple $\bar{x}$ is in the domain of f, i.e., $f(\bar{x})$ is defined; and $f(\bar{x}) \uparrow$ means that $\bar{x}$ is not in the domain of f, i.e. $f(\bar{x})$ is undefined.

[6]:-)

Example 2.8.1. Let

$$f(x) = \begin{cases} 3, & \text{if } x > 2\,; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Then f is the function whose value at x is 3 if $x > 2$, and that is undefined, otherwise.

Convention. In this chapter, the word *function* means *number-theoretic function*.

2.9 Composition and Primitive Recursion

Definition 2.9.1. Suppose that f and g are partial functions. Then $f(\bar{x}) \simeq f(\bar{y})$ means that (for the particular tuples $\bar{x}$ and $\bar{y}$), either:

a) both $f(\bar{x}) \downarrow$ and $g(\bar{y}) \downarrow$ and $f(\bar{x}) = g(\bar{y})$, or

b) both $f(\bar{x}) \uparrow$ and $g(\bar{y}) \uparrow$.

For example, $f(3) \simeq g(4)$ is a true statement if both $f(3)$ and $g(4)$ are undefined.

Exercise 2.9.2. Suppose that f is a function such that for all m,

$$f(m) \simeq f(m+1)\ .$$

Describe all possible such functions.

There are several ways of combining functions to form new functions.

Definition 2.9.3. Let $h_0, \dots, h_{k-1}$ be m-ary partial functions, and let g be a k-ary partial function. The partial function f, defined by *composition* from g and $h_0, \dots, h_{k-1}$ is that function that satisfies for each m-tuple $\bar{x} = x_0, \dots, x_{m-1}$,

$$f(\bar{x}) = \begin{cases} g(h_0(\bar{x}), \dots, h_{k-1}(\bar{x})), & \text{if } h_i(\bar{x}) \downarrow \text{ for each } i < k, \text{ and } g(h_0(\bar{x}), \dots, h_{k-1}(\bar{x})) \downarrow; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Sometimes we write

$$f(\bar{x}) \simeq g(h_0(\bar{x}), \dots, h_{k-1}(\bar{x}))$$

to indicate that f is defined by composition from g and $h_0, \dots, h_{k-1}$.

Note that if each of $g, h_0, \dots, h_{k-1}$ is total, then so is f.

Proposition 2.9.4. *If f is defined by composition from $g, h_0, \ldots, h_{k-1}$, and each of $g, h_0, \ldots, h_{k-1}$ is partial Turing computable, then so is f.*

Proof. We seek a Turing Machine M that on initial configuration

$$\begin{array}{c} \ldots 0\, 1^{x_0+1} 0 \ldots 0\, 1^{x_{m-1}+1} 0 \ldots \\ \hphantom{\ldots 0\, 1^{x_0+1} 0 \ldots 0\, 1^{x_{m-1}+1}} \wedge \hphantom{\ldots} \end{array}$$

has terminal configuration

$$\begin{array}{c} \ldots 0\, 1^{x_0+1} 0 \ldots 0\, 1^{x_{m-1}+1} 0\, 1^{g(h_0(\bar{x}),\ldots,h_{k-1}(\bar{x}))+1} 0 \ldots \\ \hphantom{\ldots 0\, 1^{x_0+1} 0 \ldots 0\, 1^{x_{m-1}+1} 0\, 1^{g(h_0(\bar{x}),\ldots,h_{k-1}(\bar{x}))+1}} \wedge \hphantom{\ldots} \end{array}$$

if $f(\bar{x}) \downarrow$,
and does not halt if $f(\bar{x}) \uparrow$. Let $\mathrm{T}_g, \mathrm{T}_{h_0}, \ldots, \mathrm{T}_{h_{k-1}}$ be Turing Machines that compute $g, h_0, \ldots, h_{k-1}$, respectively. Then the following machine computes f:

$$\begin{array}{l} \to \mathrm{T}_R \to \mathrm{T}_1 \to \mathrm{T}_\mathrm{R} \to \mathrm{T}^m_{(m+1)\mathrm{copy}} \to \mathrm{T}^m_{\mathrm{Lseek0}} \to \mathrm{T}_L \to \mathrm{T}_0 \to \mathrm{T}_{\mathrm{Rend}} \to \\ \mathrm{T}_{h_0} \to \mathrm{T}^m_{(m+1)\mathrm{copy}} \to \mathrm{T}_{h_1} \to \cdots \to \mathrm{T}^m_{(m+1)\mathrm{copy}} \to \mathrm{T}_{h_{k-1}} \to \\ \mathrm{T}_{(k+(k-1)m)\mathrm{copy}} \to \mathrm{T}_{(k+(k-2)m)\mathrm{copy}} \to \cdots \to \mathrm{T}_{k\mathrm{copy}} \to \mathrm{T}_g \to \\ \mathrm{T}_{\mathrm{fin}}\ . \end{array}$$

□

Definition 2.9.5 (Primitive Recursion without parameters)**.** Let $a \in \mathbb{N}$ and h be a 2-ary partial function. The 1-ary function f defined by *primitive recursion from a and h*, is the unique partial function f that satisfies the recursion equations:

$$\begin{aligned} f(0) &= a\ ; \\ f(m+1) &\simeq h(m, f(m))\ . \end{aligned}$$

Note that if h is total, then so is f.

Example 2.9.6. Let $h(x, y) = 2^y$ and f be defined by primitive recursion from 2 and h. Then

$$\begin{aligned} f(0) &= 2\ ; \\ f(1) &= 2^2\ ; \\ f(m) &= 2^{\cdot^{\cdot^{2}}}\ . \end{aligned}$$

Definition 2.9.7 (Primitive Recursion with parameters)**.** Let g be a k-ary partial function and h be a $k+2$-ary partial function. The function f defined by primitive recursion from g and h is the partial function that satisfies the recursion equations:

$$f(0,\bar{x}) \simeq g(\bar{x}) \ ;$$
$$f(m+1,\bar{x}) \simeq h(m,\bar{x},f(m,\bar{x})) \ .$$

Note that if g and h are total, then so is f.

As we shall see, the operation of primitive recursion, when combined with that of composition, is very powerful.

Definition 2.9.8 (Initial Functions)**.** For each $k > 0$, $n > 0$, and $i < k$, let

a) Cs^k_n be the k-ary *constant function* with constant value n; i.e.

$$\mathrm{Cs}^k_n(\bar{x}) = n$$

for every k-tuple $\bar{x}$.

b) Pr^k_i is the k-ary *projection function*,

$$\mathrm{Pr}^k_i(\bar{x}) = x_i \ ,$$

where $\bar{x} = x_0, \ldots, x_{k-1}$.

c) Similarly, Sc^k_i is the k-ary *successor function*,

$$\mathrm{Sc}^k_i(\bar{x}) = x_i + 1 \ ,$$

where $\bar{x} = x_0, \ldots, x_i, \ldots, x_{k-1}$.

The functions Cs^k_n, Pr^k_i, and Sc^k_i are called *initial functions*.

Note that there are an infinite number of initial functions.

Definition 2.9.9 (Primitive Recursive Function)**.** A function f is *primitive recursive* iff there is a finite sequence $f_0, \ldots, f_k$ of functions such that

a) $f = f_k$, and

b) each function in the list is either an initial function or is obtained from previous functions in the list by composition or primitive recursion.

Such a sequence, $f_0, \ldots, f_k$, is called a *primitive recursive sequence that constructs f*.

Exercise 2.9.10. Every primitive recursive function is total.

2.10 Lots of Primitive Recursive Functions

We show that a lot of functions are primitive recursive.

Proposition 2.10.1. *+ is primitive recursive.*

Proof. We show that the binary function + is defined by primitive recursion from the functions Pr^1_0 and Sc^3_2. We have:

$$+(0,x) = x = \mathrm{Pr}^1_0(x)\ , \tag{2.158}$$

$$+(m+1,x) = +(m,x)+1 = h(m,x,+(m,x))\ ,\ \text{where} \tag{2.159}$$

h is the 3-ary function:

$$h(m,x,u) = u+1 = \mathrm{Sc}^3_2(m,x,u)\ ,$$

so $h = \mathrm{Sc}^3_2$. Finally, the three functions

$$\mathrm{Pr}\ ,\ \mathrm{Sc}^3_2\ ,\ +$$

form a primitive recursive sequence that constructs +. □

Proposition 2.10.2 (Closure of Primitive Recursive Functions under Composition and Primitive Recursi

If f is defined by composition from g and $h_0,\ldots,h_{k-1}$, and each of g and $h_0,\ldots,h_{k-1}$ is primitive recursive, then f is primitive recursive.

b) If f is defined by primitive recursion without parameters from a and h, and h is primitive recursive, then f is primitive recursive.

c) If f is defined by primitive recursion from functions g and h, and g and h are primitive recursive, then f is primitive recursive.

Proposition 2.10.3. *$\times$ is primitive recursive.*

Definition 2.10.4.

1. $0^0 = 1$;
2. $0^x = 0$ for $x > 0$;
3. $0! = 1$.

Exercise 2.10.5. Show that the factorial function ! and the exponential function f are primitive recursive, where $f(x, y) = x^y$.

Definition 2.10.6.

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

$$\overline{\mathrm{sg}}(x) = \begin{cases} 1, & \text{if } x = 0, \\ 0, & \text{if } x > 0. \end{cases}$$

$$\mathrm{pd}(x) = \begin{cases} x - 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0. \end{cases}$$

Exercise 2.10.7. Show that sg, $\overline{\mathrm{sg}}$, and pd are primitive recursive.

The following illustrates how the projection functions are used. We will frequently use them in this manner without explicit mention.

Example 2.10.8. Suppose that $g(x, y, z)$ is primitive recursive. Then so is f, where $f(x, y, z) = g(z, y, z)$.

Proof.

$$f(x, y, z) = g(\mathrm{Pr}^3_2(x, y, z),\ \mathrm{Pr}^3_1(x, y, z), \mathrm{Pr}^3_2(x, y, z)) .$$

□

Definition 2.10.9. $\dot{-}$ is the binary function:

$$\dot{-}(x, y) = x \dot{-} y = \begin{cases} x - y, & \text{if } x > y, \\ 0, & \text{if } x \leq y. \end{cases}$$

Exercise 2.10.10. Show that $\dot{-}$ is primitive recursive.

Recall that if R is a k-ary relation on $\mathbb{N}$, then C_R, the characteristic function of R is the function:

$$C_R(\bar{x}) = \begin{cases} 1, & \text{if } \bar{x} \in R; \\ 0, & \text{if } \bar{x} \notin R. \end{cases}$$

Definition 2.10.11. A k-ary relation on $\mathbb{N}$ is primitive recursive if its characteristic function C_R is primitive recursive.

Exercise 2.10.12. Show that the binary relations =, $<$, $\leq$, and $\geq$, are primitive recursive.

Proposition 2.10.13. *If R and S are primitive recursive k-ary relations then so are*

$$R \cup S\ ,\ R \cap S\ ,\ \textit{and}\, \mathbb{N}^k \setminus R\ .$$

Definition 2.10.14. Let f be a k-ary function and $f'(\bar{x}, \bar{y}) = f(\bar{x})$. f' is called an *expansion* of f.

If f is k-ary, then f' is a function of more variables than f, whose value is determined by the values of its first k variables. There are, of course, many expansions of the same f.

Proposition 2.10.15. *If f is primitive recursive and f' is an expansion of f, then f' is primitive recursive.*

Definition 2.10.16. Let g be a total $k+1$-ary function. [a)]

$\sum_{y<m} g(y, \bar{x})$ is the $k+1$-ary total function f such that

$$f(m, \bar{x}) = \begin{cases} 0, & \text{if } m = 0; \\ g(0, \bar{x}) + g(1, \bar{x}) + \cdots + g(m-1, \bar{x}), & \text{if } m > 0. \end{cases}$$

$\prod_{y<m} g(y, \bar{x})$ is the $k+1$-ary total function f such that

$$f(m, \bar{x}) = \begin{cases} 1, & \text{if } m = 0; \\ g(0, \bar{x}) \times g(1, \bar{x}) \times \cdots \times g(m-1, \bar{x}), & \text{if } m > 0. \end{cases}$$

Proposition 2.10.17. *If g is primitive recursive then so are the functions $\sum_{y<m} g(y, \bar{x})$ and $\prod_{y<m} g(y, \bar{x})$.*

Definition 2.10.18 (Bounded Search)**.** Let g be a total $k+1$-ary function. Then $\mu\, y < m\ [g(y, \bar{x}) = 0]$ is the $k+1$-ary total function f such that

$$f(m, \bar{x}) = \begin{cases} \text{the least } y < m \text{ such that } g(y, \bar{x}) = 0, & \text{if there is such a } y, \\ m, & \text{otherwise.} \end{cases}$$

Proposition 2.10.19 (Closure Under Bounded Search)**.** *If g is primitive recursive and $f(m, \bar{x}) = \mu\, y < m\ [g(y, \bar{x}) = 0]$, then f is primitive recursive.*

Abbreviations For relations R and P,

a) $R(\bar{x})$ is an abbreviation of $(\bar{x}) \in R$;

b) $R(\bar{x})$ & $P(\bar{y})$ is an abbreviation of $(\bar{x}) \in R$ and $(\bar{y}) \in P$;

c) $R(\bar{x}) \vee P(\bar{y})$ is an abbreviation of $(\bar{x}) \in R$ or $(\bar{y}) \in P$;

d) $\neg R(\bar{x})$ is an abbreviation of $(\bar{x}) \notin R$.

Proposition 2.10.20 (Definition by Cases)**.** *Let $R_0, \ldots, R_{m-1}$ be primitive recursive k-ary relations that are mutually disjoint, and let $f_0, \ldots, f_m$ be primitive recursive k-ary functions. Let*

$$f(\bar{x}) = \begin{cases} f_0(\bar{x}), & \text{if } R_0(\bar{x}); \\ \vdots & \\ f_{m-1}(\bar{x}), & \text{if } R_{m-1}(\bar{x}), \\ f_m(\bar{x}), & \text{otherwise.} \end{cases}$$

Then f is primitive recursive.

Proposition 2.10.21. *Bounded Quantification] If R is a primitive recursive relation and P and Q satisfy*

$$P(m,\bar{x}) \iff \exists y < m \; R(y,\bar{x}) \text{ , and} \tag{2.160}$$
$$Q(m,\bar{x}) \iff \forall y < m \; R(y,\bar{x}) \tag{2.161}$$

then P and Q are primitive recursive.

2.11 Coding Finite Sequences by Integers

Definition 2.11.1. a) $|$ is the binary relation:

$$x|y \iff x \text{ divides } y \text{ .}$$

b) p_n is the $n+1$-st prime number, (so $p_0 = 2$, $p_1 = 3$, etc.).

Proposition 2.11.2. *a) $|$ is a primitive recursive relation.*

b) The set of prime numbers is a primitive recursive set.

c) p_n *is a primitive recursive function.*

Definition 2.11.3. Let $\langle\,\rangle^0 = 1$, and for $k > 0$, let

$$\langle x_0, \ldots, x_{k-1}\rangle^k = p_0^{x_0+1} \cdot p_1^{x_1+1} \cdot \cdots \cdot p_{k-1}^{x_{k-1}+1} \ .$$

Numbers of the form $\langle \bar{m}\rangle^k$ (including $\langle\,\rangle = 1$) are called *sequence numbers*.

It follows from the unique factorization theorem of arithmetic that each function $\langle\,\rangle^k$, for $k > 0$, is 1-1. We usually omit the k.

Definition 2.11.4. a) Seq is the set of sequence numbers;

b) $(s)_i = \mu z < s \neg p_i^{z+2} | s$;

c) $\mathrm{lh}(s) = \mu i < s \neg p_i | s$.

d) $s * t = s \cdot t'$, where t' is obtained from t by replacing each factor p_i^n in t by $p_{\mathrm{lh}(s)+i}^n$.

Proposition 2.11.5. *a)* $(\langle x_0, \ldots, x_{k-1}\rangle)_i = x_i$, *for* $i < k$;

b) $lh(\langle x_0, \ldots, x_{k-1}\rangle) = k$;

c) $\langle x_0, \ldots, x_{k-1}\rangle * \langle y_0, \ldots, y_{m-1}\rangle = \langle x_0, \ldots, x_{k-1}, y_0, \ldots, y_{m-1}\rangle$.
In particular, a special case of (c) is

$$\langle x_0, \ldots, x_k\rangle = \langle x_0, \ldots, x_{k-1}\rangle * \langle x_k\rangle \ .$$

Remark 2.11.6. a) If $x > 0$ then $(x)_i < x$;

b) The reason for having "+1" in the definition of $\langle\,\rangle$ is to make the definition of lh simple.

c) We write $(s)_{i,j}$ for $((s)_i)_j$, etc.

Proposition 2.11.7. *The set Seq and the functions* $(\cdot)_\cdot$, *lh, and* $*$ *are primitive recursive.*

Remark 2.11.8. Let f and g be two total 1-ary functions. Let k be the 1-ary function $k(n) = \langle f(n), g(n)\rangle$. Then if both f and g are primitive recursive so is k.

Exercise 2.11.9 (Simultaneous Recursion)**.** Let h_1 and h_2 be primitive recursive functions, and f and g be defined so that

$$\begin{aligned} f(0) &= a \ ; & g(0) &= b \ ; \\ f(n+1) &= h_1(n, f(n), g(n)); & g(n+1) &= h_2(n, f(n), g(n)) \ . \end{aligned}$$

Show that f and g are primitive recursive.

2.12 Course of Values Recursion

There are many modifications of definition by primitive recursion. One example was given in Exercise ???. In this section we describe a procedure for showing that the class of primitive recursive functions is closed under many such operations.

Definition 2.12.1. For f a k-ary total function, $\hat{f}$ is the k-ary function defined by:

$$\begin{aligned} \hat{f}(0, \bar{x}) &= \langle\,\rangle = 1 \ , \\ \hat{f}(m, \bar{x}) &= \langle f(0, \bar{x}), \dots, f(m-1, \bar{x}) \rangle \ , \text{ for } m > 0 \ . \end{aligned}$$

Thus $\hat{f}(m, \bar{x})$ codes the first m values of f. In the special case where f is 1-ary, then

$$\hat{f}(x) = \langle f(0), \dots, f(x-1) \rangle \ .$$

Lemma 2.12.2. *f is primitive recursive iff $\hat{f}$ is primitive recursive.*

Definition 2.12.3 (Course of Values Recursion)**.** Let g be a $k+2$-ary function and f be the $k+1$-ary function defined by:

$$f(m, \bar{x}) = g(m, \bar{x}, \hat{f}(m, \bar{x})) \ .$$

f is said to be defined by *course of values recursion from g*.

Example 2.12.4. Ordinary primitive recursion is a special case of course of values recursion. For example, suppose that f is defined by the primitive recursion:

$$\begin{aligned} f(0) &= a \ , \\ f(m+1) &= h(m, f(m)) \ . \end{aligned} \tag{2.162}$$

This can be rewritten as the following course of values recursion:

$$f(x) = \begin{cases} a & \text{if } x = 0; \\ h(x \dot{-} 1), (\hat{f}(x))_{x \dot{-} 1} & \text{if } x > 0. \end{cases}$$

The following important theorem shows that the converse is true.

Theorem 2.12.5 (Course of Values Recursion Theorem). *If g is primitive recursive and f is defined by course of values recursion from g, then f is primitive recursive.*

Exercise 2.12.6 (Recursion from a Double Basis). Let h be a 3-ary primitive recursive function and f be defined by

$$\begin{aligned} f(0) &= a \ ; \\ f(1) &= b \ ; \\ f(m+2) &= h(m, f(m+1), f(m)) \ . \end{aligned}$$

Show that f is primitive recursive.

The Course of Values Recursion Theorem can be used to show that many sets and relations with complex definitions are primitive recursive.

Example 2.12.7. Let $A \subset \mathbb{N}$ be the unique set[7] such that: $a \in A$ iff

$$\begin{aligned} &a = 17 \vee \exists x, y < a \ [x \in A \ \& \ y \in A \ \& \ a = \langle 5, x, y \rangle] \vee \\ &\exists s, k < a \ [\mathrm{Seq}(s) \ \& \ \mathrm{lh}(s) = k \ \& \ \forall i < k \ [(s)_i \in A] \ \& \ a = \langle 6, s \rangle] \ . \end{aligned}$$

We show that the set A is primitive recursive. By switching over to characteristic functions, we can rewrite the above expression as:

$$\begin{aligned} &a = 17 \vee \exists x, y < a \ [C_A(x) = 1 \ \& \ C_A(y) = 1 \ \& \ a = \langle 5, x, y \rangle] \vee \\ &\exists s, k < a \ [\mathrm{Seq}(s) \ \& \ \mathrm{lh}(s) = k \ \& \ \forall i < k \ [C_A((s)_i) = 1] \ \& \ a = \langle 6, s \rangle] \ . \end{aligned}$$

Let $Q(a, m)$ be the binary relation defined by:

$$\begin{aligned} &a = 17 \vee \exists x, y < a \ [(m)_x = 1 \ \& \ (m)_y = 1 \ \& \ a = \langle 5, x, y \rangle] \vee \\ &\exists s, k < a \ [\mathrm{Seq}(s) \ \& \ \mathrm{lh}(s) = k \ \& \ \forall i < k \ [(m)_{(s)_i} = 1] \ \& \ a = \langle 6, s \rangle] \end{aligned}$$

Then Q is primitive recursive, and

$$C_A(a) = 1 \iff a \in A \tag{2.163}$$

$$\iff Q(a, \hat{C}_A(a)) \ . \tag{2.164}$$

Hence $C_A(a) = C_Q(a, \hat{C}_A(a))$. So C_A is defined by course of values recursion from the primitive recursive function C_Q; hence C_A is primitive recursive, and so A is primitive recursive. □

[7] Why is there a unique set A satisfying the following condition?

2.13 Primitive Recursive Functions and Turing Computability

We show that every primitive recursive function is Turing computable.

Lemma 2.13.1. *If f is defined by primitive recursion from function(s) that are partial Turing computable, then f is partial Turing computable.*

Proof. Suppose first that f is defined by primitive recursion without parameters from a and the partial Turing computable function h. Then

$$\begin{aligned} f(0) &= a \ ; \\ f(m+1) &\simeq h(m, f(m)) \ . \end{aligned}$$

Let T_h be a Turing Machine that computes h. Then the following Turing Machine computes f.

$$\rightarrow \mathrm{T_R} \rightarrow \mathrm{T_1} \rightarrow \mathrm{T_R} \rightarrow \mathrm{T_{2copy}} \rightarrow \mathrm{T_{Lseek0}} \rightarrow \mathrm{T_L} \rightarrow \mathrm{T_0} \rightarrow \mathrm{T_{Rend}} \rightarrow$$

$$\begin{tikzcd}
\mathrm{T_R} \arrow[r] & (\mathrm{T_1} \rightarrow \mathrm{T_R})^{a+1} \arrow[r] & \mathrm{T_{2copy}} \arrow[r] & \mathrm{T_L} \arrow[r] & \mathrm{T_0} \arrow[r] & \mathrm{T_L} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \mathrm{T_R} \arrow[r] & {} \\
 & & & & & \mathrm{T_{fin}} & &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathrm{T_R} \arrow[r] & \mathrm{T_1} \arrow[r] & \mathrm{T_R} \arrow[r] & \mathrm{T_{3copy}} \arrow[r] & \mathrm{T}_h \arrow[r] & \mathrm{T_{4copy}} \arrow[r] & \mathrm{T_L} \arrow[r] & \mathrm{T_0} \arrow[r] & \mathrm{T_L} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \mathrm{T_R} \arrow[r] & \mathrm{T_{4copy}} \\
 & & & & & & & & \mathrm{T_{fin}} & &
\end{tikzcd}$$

Now suppose that that g and h are partial Turing computable functions and f is defined by the primitive recursion

$$\begin{aligned} f(0,\bar{x}) &\simeq g(\bar{x}) \ ; \\ f(m+1,\bar{x}) &\simeq h(m,\bar{x},f(m,\bar{x})) \ . \end{aligned}$$

Let

$$\begin{aligned} h'(\bar{x},m,u) &\simeq h(m,\bar{x},u) \ , \text{ and} \\ f'(\bar{x},m) &\simeq f(m,\bar{x}) \ . \end{aligned}$$

Then

$$\begin{aligned} f'(\bar{x},0) &\simeq g(\bar{x}) \\ f'(\bar{x},m+1) &\simeq h'(\bar{x},m,f'(\bar{x},m)) \ . \end{aligned}$$

If we can show that f' is Turing computable, then using the method of Example ???, so is f. Let T_g and $\mathrm{T}_{h'}$ be Turing Machines that compute g and h', respectively. Then the following machine computes f':

$$\rightarrow \mathrm{T}_{\mathrm{R}} \rightarrow \mathrm{T}_1 \rightarrow \mathrm{T}_{\mathrm{R}} \rightarrow \mathrm{T}_{\mathrm{2copy}} \rightarrow \mathrm{T}^{k}_{\mathrm{(k+3)copy}} \rightarrow \mathrm{T}^{k+1}_{\mathrm{Lseek0}} \rightarrow \mathrm{T}_{\mathrm{L}} \rightarrow$$

$$\begin{tikzcd} \mathrm{T}_0 \arrow[r] & \mathrm{T}_{\mathrm{Rend}} \arrow[r] & \mathrm{T}_g \arrow[r] & \mathrm{T}_{\mathrm{(k+2)copy}} \arrow[r] & \mathrm{T}_{\mathrm{L}} \arrow[r] & \mathrm{T}_0 \arrow[r] & \mathrm{T}_{\mathrm{L}} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \mathrm{T}_{\mathrm{R}} \arrow[r] & {} \\ & & & & & & \mathrm{T}_{\mathrm{fin}} & & \end{tikzcd}$$

$$\mathrm{T}^{k}_{\mathrm{(k+2)copy}} \rightarrow \mathrm{T}_{\mathrm{R}} \rightarrow \mathrm{T}_1 \rightarrow \mathrm{T}_{\mathrm{R}} \rightarrow \mathrm{T}_{\mathrm{(k+3)copy}} \rightarrow \mathrm{T}_h \rightarrow$$

$$\begin{tikzcd} \mathrm{T}_{\mathrm{(k+4)copy}} \arrow[r] & \mathrm{T}_{\mathrm{L}} \arrow[r, "\text{if } 1"] \arrow[d, "\text{if } 0"] & \mathrm{T}_R \arrow[r] & \mathrm{T}^{k+1}_{\mathrm{(k+4)copy}} \\ & \mathrm{T}_{\mathrm{fin}} & & \end{tikzcd}$$

□

Theorem 2.13.2. *Every primitive recursive function is Turing computable.*

2.14 Gödel Numbering of Primitive Recursive Functions

Theorem ??? says that every primitive recursive function is Turing computable. Is every total Turing computable function primitive recursive? We answer now a slightly easier question:

Is every effectively computable total function primitive recursive?

In order to answer this question we use our coding of finite sequences by integers to code primitive recursive functions by integers. This technique is known as 'Gödel numbering.'

Definition 2.14.1. We assign to each primitive recursive function f an integer name called its Gödel number. First we assign Gödel numbers to initial functions.

INITIAL FUNCTION	GÖDEL NUMBER
Cs_n^k	$\langle 0, k, 0, n\rangle$
Pr_i^k	$\langle 0, k, 1, i\rangle$
Sc_i^k	$\langle 0, k, 2, i\rangle$

Next we extend the Gödel numbering to include functions defined by composition and primitive recursion from functions to which Gödel numbers have already been assigned.

i) If f is an m-ary function defined by composition from the primitive recursive functions g and $h_0, \dots, h_{k-1}$ with Gödel numbers b and $c_0, \dots, c_{k-1}$, respectively, then $\langle 1, m, b, \langle c_0, \dots, c_{k-1}\rangle\rangle$ is a Gödel number of f.

ii) If f is a 1-ary function defined by primitive recursion from a and the 2-ary primitive recursive function h with Gödel number b, then $\langle 2, 1, a, b\rangle$ is a Gödel number of f.

iii) If f is a $k+1$-ary function defined by primitive recursion from the k-ary primitive recursive function g and the $k+2$-ary primitive recursive function h with Gödel numbers b and c, respectively, then $\langle 2, k+1, b, c\rangle$ is a Gödel number of f.

Remark 2.14.2. Note that every primitive recursive function has a Gödel number, and in fact there are an infinite number of Gödel numbers for each primitive recursive function.

We first look at the set of Gödel numbers of primitive recursive functions.

Definition 2.14.3. Pri is the set of Gödel numbers of primitive recursive functions.

Lemma 2.14.4. *Pri is a primitive recursive set.*

Proof. We use the same reasoning that was used in the proof of Example 2.12.7 $a \in \text{Pri}$ iff

$$\exists i,k,n < a\ [0 < k \ \&\ a = \langle 0,k,0,n\rangle \vee i < k \ \&\ [a = \langle 0,k,1,i\rangle \vee a = \langle 0,k,2,i\rangle]]\vee$$

$$\exists b,c,k,m < a\ [a = \langle 1,m,b,c\rangle \ \&\ b \in \text{ Pri} \ \&\ (b)_1 = k \ \&\ \text{Seq}(c) \ \&\ \text{lh}(c) = k \ \&$$

$$\forall i < k\ [(c)_i \in \text{Pri} \ \&\ (c)_{i,1} = m]]\vee$$
$$\exists b,c < a\ [a = \langle 2,1,b,c\rangle \ \&\ c \in \text{Pri} \ \&\ (c)_1 = 2]\vee$$
$$\exists b,c,k < a\ [a = \langle 2,k+1,b,c\rangle \ \&\ b \in \text{ Pri} \ \&\ c \in \text{ Pri} \ \&\ (b)_1 = k \ \&\ (c)_1 = k+2]\ .$$

□

Remark 2.14.5. Our Gödel numbering of primitive recursive functions has the following important feature:

Given an integer a, we can effectively tell whether or not a is the Gödel number of a primitive recursive function (since Pri is primitive recursive) and if it is, we can effectively recover from a the primitive recursive function that a is a name of (and then of course we can effectively evaluate that function at any arguments we want).

Definition 2.14.6. For $a \in \text{Pri}$, let ϕ_a be the primitive recursive function with Gödel number a.

Example 2.14.7. Let $a = \langle 2, 2, \langle 0, 1, 1, 0\rangle, \langle 0, 3, 2, 2\rangle\rangle$. Then

$$\begin{aligned}\phi_a(1,1) &= \phi_{\langle 0,3,2,2\rangle}(0,1,\phi_a(0,1)) && (2.165)\\ &= \phi_a(0,1)+1 && (2.166)\\ &= \text{Pr}^1_0(1)+1 && (2.167)\\ &= 1+1 && (2.168)\\ &= 2\ . && (2.169)\end{aligned}$$

What familiar function is ϕ_a?

Theorem 2.14.8 (Informal Theorem)**.** *There is an effectively computable total function that is not primitive recursive.*

Proof. Let

$$f(a,x) = \begin{cases} \phi_a(x), & \text{if } a \in \text{Pri} \;\&\; (a)_1 = 1; \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, f is effectively computable. Let

$$g(x) = f(x,x) + 1 \text{ for all } x \ .$$

Clearly, g is also effectively computable. If g is primitive recursive, then it has a Gödel number. That is, for some a,

$$g(x) = f(a,x) \text{ for all } x \ .$$

But then,

$$f(a,a) = g(a) = f(a,a) + 1 \ ,$$

which is impossible. Hence g is not primitive recursive. □

The class of primitive recursive functions, although a very large class that contains all the functions (such as exponentiation and the factorial function) that appear in elementary number theory, is a proper subset of the class of effectively computable functions. At this point is natural to add new functions to the class of primitive recursive functions in an attempt to get all effectively computable functions. One way of doing this is to add the function g, defined above, to the class of initial functions. More generally:

Definition 2.14.9. Let g be a total function (of any number of variables). A function f is *primitive recursive in g* if there is a finite sequence

$$f_0, \ldots f_k$$

of functions such that:

a) $f = f_k$, and

b) each f_i in the list is either an initial function, or is the function g, or is obtained from previous functions in the list by either composition or primitive recursion.

Question 2.14.10. Let g be a total, effectively computable function that is not primitive recursive (such as the function g defined in the proof of Theorem ???. Is every total Turing computable function primitive recursive in f?

Answer: *No*. For we can repeat the argument of Theorem 2.14.8. We define Pri as before, except add a new Gödel number $\langle 0, 1, 3\rangle$ (assuming that g is a function of one variable) as a name of g. The new set Pri is still primitive recursive. Then define g_1 just like the g in the proof of Theorem 2.14.8. g_1 is still effectively computable and, by repeating the reasoning of Theorem 2.14.8, g_1 is not primitive recursive in g.

We can now repeat the process, getting a g_2 that is effectively computable but not primitive recursive in g_1, etc. This points up a basic difficulty in getting a really nice, workable, mathematical definition of the set of effectively computable, total functions. The remedy is to work from the beginning with partial functions. We do this in the next section.

2.15 Unbounded Search and Partial Recursive Functions

Suppose that g is a total 2-ary function. Let

$$f(x) = \begin{cases} \text{the least } y \text{ such that } g(y,x) = 0, & \text{if there is such a } y; \\ 0, & \text{otherwise.} \end{cases}$$

Even if g is effectively computable there is no reason to believe that f must be computable also.

Let's try again. Let g be a total 2-ary function, and let

$$f(x) \simeq \begin{cases} \text{the least } y \text{ such that } g(y,x) = 0, & \text{if there is such a } y; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Now if g is effectively computable then so is f, except that now f might no longer be a total function. This form of *unbounded search* leads outside the set of total functions. Because of this, it is natural to apply this operation to functions g that are not total. So now let g be partial, and

$$f(x) \simeq \begin{cases} \text{the least } y \text{ such that } g(y,x) = 0, & \text{if there is such a } y; \\ \uparrow, & \text{otherwise.} \end{cases}$$

Unfortunately, even if g is partial effectively computable, there is again no reason to believe that f is partial effectively computable. (Why?)

Definition 2.15.1 (Unbounded Search)**.**

a) Let g be a $k+1$-ary partial function. We define $\mu\, y\ [g(y,\bar{x})=0]$ to be the partial function of $\bar{x}$ such that

$$\mu y[g(y,\bar{x})=0] \simeq \begin{cases} y, & \text{if } g(y,\bar{x})=0\ \&\forall z<y\ [g(z,\bar{x}) \text{ is defined and not equal to } 0]; \\ \uparrow, & \text{if there is no such } y. \end{cases}$$

b) For a relation R, we write $\mu y\, R(y,\bar{x})$ as an abbreviation of the partial function $\mu y[\overline{\mathrm{sg}}(C_R(y,\bar{x}))=0]$.

It s easy to see that if g is a partial effectively computable function then so is the partial function $\mu\, y[g(y,\bar{x})=0]$. A mathematical counterpart to this is:

Proposition 2.15.2. *If g is a $k+1$-ary partial Turing computable function, then $\mu y[g(y,\bar{x})=0]$ is a k-ary partial Turing computable function.*

Proof. Let $g'(\bar{x},y)\simeq g(y,\bar{x})$. Then g' is partial Turing computable (why?) and for all $\bar{x}$,

$$\mu y[[g'(\bar{x},y)=0] \simeq \mu y\ [g(y,\bar{x})=0]\ .$$

So it suffices to show that $\mu y[g'(\bar{x},y)=0]$ is partial Turing computable. The following Turing machine computes $\mu y\ [g'(\bar{x},y)=0]$:

$$\mathrm{T_R} \to \mathrm{T_1} \to \mathrm{T_R} \to \mathrm{T}_{g'} \to \mathrm{T_L} \to \mathrm{T_0} \to \mathrm{T_L} \xrightarrow{\text{if } 1} \mathrm{T_0} \to \mathrm{T_L}$$

□

We now ‘add’ the operation of unbounded search to the operations of composition and primitive recursion.

Definition 2.15.3. The function f is *partial recursive* if there is a finite sequence

$$f_o,\ldots,f_k$$

of partial functions such that:

a) $f=f_k$, and

b) each function in the list is either an initial function or is obtained from previous functions in the list by composition, or primitive recursion, or unbounded search.

Definition 2.15.4. A function is *recursive* if it is partial recursive and total.

Theorem 2.15.5.

a) Every partial recursive function is partial Turing computable.

b) Every recursive function is Turing computable.

Definition 2.15.6. Let R be a k-ary relation on $\mathbb{N}$.

a) R is *recursively enumerable* if R is equal to the domain of some partial recursive function.

b) R is *recursive* if the characteristic function C_R is a recursive function.

Corollary 2.15.7. *For every relation R on $\mathbb{N}$:*

a) If R is recursive then R is Turing computable.

b) If R is recursively enumerable then R is Turing semi-computable.

2.16 Summary

Summarizing what we have shown so far

a) For every number-theoretic function f:

$$\begin{aligned} f \text{ partial recursive} &\Longrightarrow f \text{ partial Turing computable} && (2.170)\\ &\Longrightarrow f \text{ is partial effectively computable.} && (2.171)\end{aligned}$$

b) For every number-theoretic function f:

$$\begin{aligned} f \text{ recursive} &\Longrightarrow f \text{ is Turing computable} && (2.172)\\ &\Longrightarrow f \text{ is effectively computable.} && (2.173)\end{aligned}$$

c) For every relation R on $\mathbb{N}$:

$$\begin{aligned} R \text{ recursively enumerable} &\Longrightarrow R \text{ is Turing semi-computable} && (2.174)\\ &\Longrightarrow R \text{ is effectively enumerable.} && (2.175)\end{aligned}$$

d) For every relation R on $\mathbb{N}$:

$$\begin{aligned} R \text{ recursive} &\Longrightarrow R \text{ is Turing computable} && (2.176)\\ &\Longrightarrow R \text{ is effectively decidable.} && (2.177)\end{aligned}$$

What about the converses? Do the arrows go in the other direction? In the 1930's Alonzo Church proposed the following:

Church's Thesis:

a) Every effectively computable partial function is partial recursive. And so:

b) Every effectively computable total function is recursive.

c) Every effectively enumerable relation is recursively enumerable. And so:

d) Every effectively decidable relation is recursive.

Note that Church's Thesis is not directly subject to mathematical proof since it asserts the equivalence between the informal notion of effectively computable partial function and the mathematical notion of partial recursive function. On the other hand, if false, it could be refuted by providing a counterexample. The best one can do (short of providing a counterexample) is to provide evidence for the truth of Church's Thesis. This evidence is basically of three kinds:

a) All the properties that informal reasoning shows hold for effectively computable functions, (respectively, effectively enumerable relations, and Turing computable relations) are mathematical theorems about partial recursive functions (respectively, recursively enumerable relations, and recursive relations). As an example, see ???, below.

b) In the 80+ years since Church's Thesis was proposed, despite intensive scrutiny, nobody has been able to find a counterexample or even suggest a plausible approach to finding one.

c) There have been a number of attempts to provide a mathematical characterization of the class of effectively computable functions by different mathematicians, using widely different approaches. Without exception, all of the mathematical classes turn out to be equal to the class of partial recursive functions.

Much of the material that follows may be regarded as evidence for Church's Thesis. In particular, we have seen two attempts at a mathematical characterization of effectively computable function. The first is via Turing Machines; the second gives the class of partial recursive functions. One of our objectives is to show that a partial function is Turing computable iff it is partial recursive. In view of (c) above, this may be regarded as evidence supporting Church's Thesis.[8]

Exercise 2.16.1. Show that if the relation R is recursive then both R and its complement are recursively enumerable. (Do not use Church's Thesis.)

Do you think that the converse is true? What does Church's Thesis lead you to believe?

Exercise 2.16.2. Show that if P is a $k+1$-ary recursive relation and

$$R(\bar{x}) \iff \exists y\, P(y, \bar{x})$$

then R is recursively enumerable.

Exercise 2.16.3. Let $f(\bar{x}) \simeq \mu y\ R(y, \bar{x})$. Show that

$$f(\bar{x})\downarrow \ \text{ iff } \ \exists y\ R(y, \bar{x})\ .$$

Exercise 2.16.4. Let R be recursive, and

$$f(\bar{x}) \simeq \begin{cases} 1 & \text{if } R(\bar{x}); \\ \uparrow & \text{otherwise.} \end{cases}$$

Show that f is partial recursive.

Question 2.16.5. If R is a k-ary recursively enumerable relation does there necessarily exist a recursive $k+1$-ary relation P such that

$$R(\bar{x}) \iff \exists y\, P(y, \bar{x})\ ?$$

Question 2.16.6. If R is a $k+1$-ary recursively enumerable relation and

$$Q(\bar{s}) \iff \exists y\, R(y, \bar{x})$$

is Q necessarily recursively enumerable? What does Church's Thesis lead you to believe? (*Hint:* See Exercise ???)

Question 2.16.7. Can every partial recursive function be extended to a recursive function? More precisely, if f is a partial recursive 1-ary function is there necessarily a *recursive* function g such that $g(x) = f(x)$ whenever $f(x)\downarrow$?

[8] 'Church's Thesis' is also referred to as the 'Church-Turing' Thesis.

2.17 Kleene Normal Form

We know that every partial recursive function is partial Turing computable. Next we turn our attention to the converse. We wish to show that every partial Turing computable function is partial recursive. Suppose, for example, that f is a 1-ary partial Turing computable function. The following is a brief outline of the proof that f is partial recursive. Let M be a Turing machine that computes f. Then for all x, y, $f(x) = y$ iff there exists a computation $C_0, \ldots, C_k$ of M that on initial configuration

$$q_0 \ : \ \ldots 01^{x+1}00^{\infty}$$

has terminal configuration of the form

$$q \ : \ \ldots 01^{x+1}01^{y+1}00^{\infty}$$

Rewriting this we have:

$$\begin{aligned} f(x) = y \iff \exists C \exists k [(\& C = (C_0, \ldots, C_k) \text{ is a computation of M} \\ \& \ (C_0 \text{ `looks like' } q_0 \ \ \ldots 01^{x+1}00^{\infty}) \quad (2.178) \\ \& (C_k \grave{\text{l}}\text{ooks like' } q : \ \ldots 01^{x+1}01^{y+1}00^{\infty})] \quad (2.179) \end{aligned}$$

Let $Q(x, y, C)$ be the $\exists k[\ldots]$ part of the above expression. Then

$$f(x) = y \iff \exists C Q(x, y, C) \ .$$

The idea behind the proof is to assign Gödel numbers to Turing machines, tape descriptions, and computations of Turing machines, in such a way that from the relation Q we get a relation Q of *numbers* (i.e. $Q \subseteq \mathbb{N}^3$) so that

$$\begin{aligned} f(x) = y &\iff \exists u Q(x, y, u) && (2.180) \\ &\iff \exists w [Q(x, (w)_0, (w)_1) \& (w)_0 = y]. && (2.181) \end{aligned}$$

We then have

$$f(x) \simeq (\mu w Q(x, (w)_0, (w)_1))_0$$

Note that f is defined by composition from the function $(\cdot)_.$, and a function defined by unbounded search from Q. Hence, if we can show that the Gödel numbering can be done in such a way that makes Q recursive, then we have the desired result that f is partial recursive.

When we actually go through the process of Gödel numbering we do not have to have a different Q for each partial Turing computable function f. Instead it is just as simple to do it simultaneously for all f by simply having an extra variable in Q to stand for the Gödel number of M.

2.18 Gödel Numbering of Turing Machines and Computations

Gödel Numbering of Turing Machines

We assign Gödel numbers as names for Turing machines on alphabet $\{1\}$ with 0 as the blank. A Turing machine is described completely by its ordered list of quadruples (which are the instructions of the machine). We begin by assigning Gödel numbers to the components of the quadruples.

SYMBOL	GOEDEL NUMBER
q_i	$\langle 0, i\rangle$
0	0
1	1
$\Leftarrow$	$\langle 1, 0\rangle$
$\Rightarrow$	$\langle 1, 1\rangle$

a) If a is one of the above symbols, we write ${}^{\#}a$ for its Gödel number. So ${}^{\#}q_i = \langle 0, i\rangle$, ${}^{\#}\Leftarrow\, = \langle 1, 0\rangle$, etc.

b) Continuing, we assign Gödel numbers to *quadruples* as follows:

$$ {}^{\#}(q_i, a, S, q_j) = \langle {}^{\#}q_i, {}^{\#}a, {}^{\#}S, {}^{\#}q_j\rangle\ , $$

where a is either 0 or 1, and $S \in \{0, 1, \Leftarrow, \Rightarrow\}$.

c) If $I_1, \ldots, I_k$ is the ordered list of instructions of the Turing machine M, then $\langle {}^{\#}I_1, \ldots, {}^{\#}I_k\rangle$ is the Gödel number of M.

d) Let ${}^{\#}T$ be the set of Gödel numbers of Turing machines.

Lemma 2.18.1. *${}^{\#}T$ is primitive recursive.*

Proof. $a \in {}^{\#}T$ iff $\mathrm{Seq}(a)$ & $\mathrm{lh}(a) \geq 1$ &

$$
\begin{aligned}
\forall i < \mathrm{lh}(a)\ \big[&\mathrm{Seq}((a)_i)\ \&\ \mathrm{lh}((a)_i) = 4\ \& \\
&\exists j, k < a\ [(a)_{i,0} = \langle 0, j\rangle\ \&\ (a)_{i,3} = \langle 0, k\rangle]\ \& \\
&\exists j < 2[(a)_{i,1} = j]\ \& \\
&\exists j < 2\ [(a)_{i,2} = j \vee (a)_{i,2} = \langle 1, j\rangle]\ \& \\
&\forall i' < \mathrm{lh}(a)\ [(a)_{i,0} = (a)_{i',0}\ \&\ (a)_{i,1} = (a)_{i',1}] \Longrightarrow \\
&\quad (a)_{i,2} = (a)_{i',2}\ \&\ (a)_{i,3} = (a)_{i',3}\big]
\end{aligned}
$$

□

Gödel Numbering of Tape Descriptions

Gödel numbers $^{\#}z$ are assigned to numbers $z \in \mathbb{Z}$ as follows:

$$\begin{array}{llllllllll}\mathbb{Z}: \dots -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & \dots \\ \mathbb{N}: \dots \ \ 7 & 5 & 3 & 1 & 0 & 2 & 4 & 6 & 8 & \dots\end{array}$$

Note that:

$$^{\#}z = \begin{cases} 2z, & \text{if } z \geq 0\ ; \\ -2z-1, & \text{if } z < 0\ . \end{cases}$$

Next Gödel numbers are assigned to tape descriptions. Recall that a tape description is a function $F\colon \mathbb{Z} \longrightarrow \{0,1\}$ such that $F(i) = 0$ except for a finite number of 1's.

Definition 2.18.2. For F a tape description, let f_F (abbreviated$f$)be the function:

$$\begin{aligned} f(0) &= F(0) \\ f(1) &= F(-1) \\ f(2) &= F(1) \end{aligned}$$

and in general, for $n \in \mathbb{N}$,

$$f(2n) = F(n)\ , \text{ and} \tag{2.182}$$

$$f(2n+1) = F(-(n+1))\ , \tag{2.183}$$

so $f(^{\#}z) = F(z)$ for $z \in \mathbb{Z}$.

Note that $f(n) = 0$ for all but finitely many n. For F a tape description, we define $^{\#}F$ (the Gödel number of F) by:

$$^{\#}F = p_0^{f(0)} p_1^{f(1)} \cdots \tag{2.184}$$

$$= \prod_{i=0}^{\infty} p_i^{f(i)}\ . \tag{2.185}$$

All but a finite number of exponents are 0, so this definition makes sense.

Example 2.18.3.

$$
\begin{array}{rcl}
z & : & \quad -2 \ -1 \ 0 \ 1 \ 2 \ 3 \qquad (2.186)\\
F & : & {}^{\infty}0 \ 0 \quad 1 \ 0 \ 0 \ 1 \ 1 \quad 0 \quad 0^{\infty} \qquad (2.187)\\
{}^{\#}z & : & \quad 3 \quad 1 \ 0 \ 2 \ 4 \ 6 \qquad (2.188)
\end{array}
$$

so $f(0) = 0$, $f(1) = 1$, $f(2) = 0$, $f(3) = 0$, $f(4) = 1$, $f(5) = 0$, $f(6) = 1$, and $f(n) = 0$ for $n > 6$, and hence

$$
\begin{aligned}
{}^{\#}F &= p_0^{f(0)} p_1^{f(1)} p_2^{f(2)} p_3^{f(3)} p_4^{f(4)} p_5^{f(5)} p_6^{f(6)} \cdots && (2.189)\\
&= 2^0 \cdot 3^1 \cdot 5^0 \cdot 7^0 \cdot 11^1 \cdot 13^0 \cdot 17^1 && (2.190)\\
&= 3 \cdot 11 \cdot 17 \ . && (2.191)
\end{aligned}
$$

Remark 2.18.4.

a) m is the Gödel number of a tape description iff

$$m > 0 \ \& \ \forall i < m \ [(m)_i = 0] \ ,$$

so the set of Gödel numbers of tape descriptions is a primitive recursive set.

b) Let exp be the primitive recursive function:

$$\exp(m, n) = \mu t < m \ \neg p_n^{t+1} | m \ .$$

Note that if m is the Gödel number of the tape description F and $n = {}^{\#}z$, then

$$
\begin{aligned}
F(z) &= f({}^{\#}z)\\
&= \text{the exponent of } p_n \text{ in the prime factorization of } m(= {}^{\#}F)\\
&= \exp(m, n) \ .
\end{aligned}
$$

Gödel Numbering of Computations

Definition 2.18.5.

a) A *complete configuration* is a quadruple (M, F, q, z), where M is a Turing machine and (F, q, z) is a configuration of M.

b) Let C be the set of complete configurations

c) Gödel numbers are assigned to complete configurations as follows:

$$^{\#}(\mathrm{M}, F, q, z) = \langle {}^{\#}\boldsymbol{M}, {}^{\#}F, {}^{\#}q, {}^{\#}z \rangle \ .$$

d) Let $^{\#}\mathcal{C}$ be the set of Gödel numbers of members of C.

Lemma 2.18.6. *$^{\#}\mathcal{C}$ is primitive recursive.*

Proof. $a \in^{\#} C$ iff

$$\begin{aligned}
&\mathrm{Seq}(a) \ \&\ \mathrm{lh}(a) = 4 \ \& \\
&(a)_0 \text{ is the Gödel number of a Turing machine M } \& \\
&(a)_1 \text{ is the Gödel number of a tape description } \& \\
&(a)_2 \text{ is the Gödel number of a state of M}
\end{aligned}$$

iff

$$\begin{aligned}
&\mathrm{Seq}(a) \ \&\ \mathrm{lh}(a) = 4 \ \& \\
&(a)_0 \in^{\#} T \ \& \\
&(a)_1 > 0 \ \&\forall i < (a)_1 \ [(a)_{1,i} = 0] \ \& \\
&\exists i < \mathrm{lh}((a)_0) \ [(a)_2 = (a)_{0,i,0} \lor (a)_2 = (a)_{0,i,3}] \ .
\end{aligned}$$

□

The next goal is to assign Gödel numbers to computations. During a computation, the numbers on the tape change; that is, the tape description changes. The following shows how a change in the tape corresponds to a change in the Gödel number of the tape description.

Definition 2.18.7. R_1 is the 4-ary relation on $\mathbb{N}$ such that $R_1(a, n, m, b)$ iff

$$\begin{aligned}
&[a =^{\#} F \text{ for some tape description} F] \ \& \\
&[n =^{\#} z \text{ for some } z \in \mathbb{Z}] \ \& \\
&[m = 0 \lor m = 1] \ \& \\
&[b =^{\#} F_z^m] \ .
\end{aligned}$$

Lemma 2.18.8. *1.8. R_1 is primitive recursive.*

Proof. $R_1(a, n, m, b)$ iff

$$\begin{aligned}
&a > 0 \;\&\; \forall i < a\ [(a)_i = 0] \;\& \\
&[m = 0 \vee m = 1] \;\& \\
&b = \left[\frac{a}{p_n^{\exp(a,n)}}\right] \cdot_n^m \ .
\end{aligned}$$

To see that R_1 is primitive recursive, observe that the last line above can be rewritten as

$$b \cdot p_n^{\exp(a,n)} = a \cdot p_n^m \ ,$$

which is a primitive recursive relation in b, a, m, and n. □

Definition 2.18.9. Let f_1 and f_2 be the total 1-ary functions such that for all $z \in \mathbb{Z}$,

$$\begin{aligned}
f_1(\overset{\#}{z}) &= {}^{\#}(z+1) \\
f_2(\overset{\#}{z}) &= {}^{\#}(z-1) \ .
\end{aligned}$$

Exercise 2.18.10. Show that f_1 and f_2 are primitive recursive by finding for $n \in \mathbb{N}$ the values of $f_1(n)$ and $f_2(n)$.

At this point you should review (Definitions ?, ? on p. 32?) the definition of a computation of a Turing machine. We now assign Gödel numbers to computations.

Definition 2.18.11. ${}^{\#}\vdash$ is the 2-ary relation such that $x \ {}^{\#}\vdash \ y$ iff

> x is the Gödel number of a complete configuration (M, F, q, z), and
> y is the Gödel number of a complete configuration (M, F', q', z'), and
> $(F, q, z) \vdash_{\mathrm{M}} (F', q', z')$.

Theorem 2.18.12. ${}^{\#}\vdash$ *is primitive recursive.*

Proof. $x \ {}^{\#}{\vdash}\ y$ iff

> x is the Gödel number of a complete configuration (M, F, q, z), and
> y is the Gödel number of a complete configuration (M, F', q', z'), and
> there is an instruction (say the $i+1$-st instruction) of M such that
> $(x)_{0,i,0} =^{\#} q \ (= (x)_2)$, and
> $(x)_{0,i,1} = F(z) \ (= \exp((x)_1, (x)_3)$ (this says
> the $i+1$-st instruction begins $(q, F(z), \ldots))$
> and one of the following holds:

a) $(x)_{0,i,2} = 0 \ \&\ R_1((x)_1, (x)_3, 0, (y)_1) \ \&\ (y)_2 = (x)_{0,i,3} \ \&\ (y)_3 = (x)_3$

b) $(x)_{0,i,2} = 1 \ \&\ R_1((x)_1, (x)_3, 1, (y)_1) \ \&\ (y)_2 = (x)_{0,i,3} \ \&\ (y)_3 = (x)_3$

c) $(x)_{0,i,2} = \ {}^{\#}{\Leftarrow} \ \&\ (y)_1 = (x)_1 \ \&\ (y)_2 = (x)_{0,i,3} \ \&\ (y)_3 = f_2((x)_3)$

d) $(x)_{0,i,2} = \ {}^{\#}{\Rightarrow} \ \&\ (y)_1 = (x)_1 \ \&\ (y)_2 = (x)_{0,i,3} \ \&\ (y)_3 = f_1((x)_3)$.

Line (a) says:

> the $i+1$-st instruction of $\boldsymbol{M}$ is $(q, F(z), 0, q')$ and$z' = z$ and $F' = F_z^0$.

You can figure out for yourself what the others say! Let $Q(i, x, y)$ be the disjunction of (a)-(d). Then Q is primitive recursive, and $x \ {}^{\#}{\vdash}\ y$ iff

$$x \in^{\#} C \ \&\ y \in^{\#} C \ \&\ (x)_0 = (y)_0 \ \&$$
$$\exists i < \mathrm{lh}((x)_0 \ [(x)_{0,i,0} = (x)_2 \ \&\ (x)_{0,i,1} = \exp((x)_1, (x)_3) \ \&\ Q(i, x, y)] \ .$$

It follows that ${}^{\#}{\vdash}$ is primitive recursive. □

Definition 2.18.13.

a) A *complete computation* is a sequence

$$(\mathrm{M}, F_0, q_0, z_0), \ldots, (\mathrm{M}, F_m, q_m, z_m)$$

such that

$$(F_0, q_0, z_0), \ldots, (F_m, q_m, z_m)$$

is a computation of the Turing machine M. The Gödel number of such a complete computation is

$$\langle {}^{\#}(\mathrm{M}, F_0, q_0, z_0), \ldots, {}^{\#}(\boldsymbol{M}, F_m, q_m, z_m) \rangle \ .$$

b) Comp is the set of Gödel numbers of complete computations.

Lemma 2.18.14. *Comp is primitive recursive.*

Proof. Comp(x) iff Seq(x) & lh$(x) \geq 1$&

$$\forall i < \mathrm{lh}(x)\ [(x)_i \in^{\#} C]\ \&$$
$$(x)_{0,0,0,0} = (x)_{0,2}\ \&\ \forall i < \mathrm{lh}(x)\dot{-}1\ [(x)_i\ {}^{\#}\vdash (x)_{i+1}]\ \&$$
$$\neg\exists i < \mathrm{lh}(x)_{0,0})\ [(x)_{0,0,i,0} = (x)_{\mathrm{lh}(x)\dot{-}1,2}\ \ \&\ (x)_{0,0,i,1} = \exp((x)_{\mathrm{lh}(x)\dot{-}1,1}, (x)_{\mathrm{lh}(x)\dot{-}1,3})]$$

The last line above says that (F_m, q_m, z_m) is a terminal configuration. To help see this note that

$$\begin{aligned} {}^{\#}q_m &= (x)_{\mathrm{lh}(x)\dot{-}1,2} \\ F_m(z_m) &= \exp((x)_{\mathrm{lh}(x)\dot{-}1,1}, (x)_{\mathrm{lh}(x)\dot{-}1,3})\ . \end{aligned}$$

□

Gödel Numbering of Computations of Partial Turing computable Functions

A computation by a Turing machine of the value of a function f begins with an input of a certain form. We need to look at the Gödel numbers of such initial tape descriptions.

Definition 2.18.15. a) If h is a finite sequence of 0's and 1's, then ${}^{\#}h$ is the integer that codes h. For example, the code of the finite sequence 01101 of 0's and 1's is $\langle 0, 1, 1, 0, 1\rangle$.

b) For each $m \in \mathbb{N}$, f_3^m is the m-ary function such that for all x,

$$f_3^m(x_0, \ldots, x_{m-1}) = {}^{\#}(1^{x_0+1}0\ldots01^{x_{m-1}+1})\ .$$

Exercise 2.18.16. Show that for each $m > 0$, f_3^m is a primitive recursive function.

Exercise 2.18.17. If h and k are finite sequences of 0's and 1's, and hk is the concatenation of h and k, then ${}^{\#}(hk) = {}^{\#}h * {}^{\#}k$.

Recall what it means for a word to lie on a tape F beginning at z and ending at z' (see Definition ?? on page ??).

Definition 2.18.18. R_2 is the 4-ary relation: $R_2(x, y, m, n)$ iff

x is the Gödel number of a tape description F, and

y is the Gödel number of a finite sequence h of 0's and 1's, and

for some $z, z' \in \mathbb{Z}$ $[m = {}^{\#}z$ & $n = {}^{\#}z'$ & h lies on F beginning at z and ending at $z']$.

Lemma 2.18.19. *R_2 is primitive recursive.*

Proof. $R_2(x, y, m, n)$ iff

1. x is the Gödel number of some tape description F &
2. $\mathrm{Seq}(y)$ & $\forall i < \mathrm{lh}(y)\ [(y)_i = 0 \vee (y)_i = 1]$ &
3. $\exists u\ \big[\mathrm{Seq}(u)$ & $\mathrm{lh}(u) = \mathrm{lh}(y)$ & $(u)_0 = {}^{\#}z$ & $(u)_{\mathrm{lh}(u) \dot{-} 1} = {}^{\#}z'$ &
4. $\forall i < \mathrm{lh}(u)\ [(u)_{i+1} = {}^{\#}(z + 1)$ &
5. $\exp(x, (u)_i) = (y)_i]\big]$

Then R_2 is primitive recursive since the last three lines above can be written as

$$\begin{aligned}\exists u < \big(p_{\mathrm{lh}(y)}^{(m+n+1)}\big)^{\mathrm{lh}(y)}\ \big[&\mathrm{Seq}(u) \ \&\ \mathrm{lh}(u) \ \&\ \mathrm{lh}(u) = \mathrm{lh}(y) \ \& \\ &(u)_0 = m \ \&\ (u)_{\mathrm{lh}(u) \dot{-} 1} = n \ \& \\ &\forall i < \mathrm{lh}(u) \dot{-} 1\ [(u)_{i+1} = f_1((u)_i) \ \& \\ &(y)_i = \exp(x, (u)_i)\big]\ .\end{aligned}$$

□

Lemma 2.18.20. *Let $m > 0$. There is a primitive recursive $m+3$-ary relation Q_m such that for every m-ary partial Turing computable function f, there is some e such that for all $\bar{x} = x_0, \ldots, x_{m-1}$ and y,*

$$f(\bar{x}) = y \iff \exists u\ Q_m(e, \bar{x}, y, u)\ .$$

Proof. Let f be an m-ary partial Turing computable function, and let M be a Turing machine that computes f. Let e be the Gödel number of M. Then $f(\bar{x}) = y$ iff

$\exists u$ $\big[u$ is the Gödel number of a complete computation

$(\mathrm{M}, F_0, q_0, z_0), \dots, (\boldsymbol{M}, F_n, q_n, z_n)$

such that $01^{x_0+1}0\dots01^{x_{m-1}+1}$ lies on F_0 ending at -1 and F_0 is 0 otherwise, and

$z_0 = 0$, and

$1^{y+1}0$ lies on the tape F_n beginning at position 1 and ending at $(y+2)\big]$

iff

$$\exists u \Big[e \in^{\#} T \ \&\ \mathrm{Comp}(u) \ \&\ (u)_{0,0} = e \ \&\ (u)_{0,3} = 0 \ \&$$

$$\exists s < u \ \big[R_2((u)_{0,1}, \langle 0\rangle * f_3^m(x_0, \dots, x_{m-1}), s, 1) \ \&$$

$$\forall t < (u)_{0,1} \ [\{(t \text{ odd} \ \&\ t > s) \vee t \text{ even}\} \Longrightarrow$$

$$\exp((u)_{0,1}, t) = 0]\big] \ \&$$

$$R_2((u)_{\mathrm{lh}(u)\dot{-}1,1}, f_3^1(y) * \langle 0\rangle, {}^{\#}1, {}^{\#}(y+2))\Big]$$

iff

$\exists u \ Q_m(e, \bar{x}, y, u)$,

where Q_m is the relation defined between the above braces $[\dots]$. □

Theorem 2.18.21 (Kleene Normal Form)**.** *There is a primitive recursive 1-ary function U, and for each $m > 0$ there is a primitive recursive $m+2$-ary relation T_m such that for every m-ary partial Turing computable function f there is some $e \in \mathbb{N}$ such that for all $\bar{x} = x_0, \dots, x_{m-1}$,*

$$f(\bar{x}) \simeq U(\mu y \ T_m(e, \bar{x}, y)) .$$

Proof. Let T_m be the following primitive recursive relation, and U be the following primitive recursive function.

$$T_m(e, \bar{x}, y) \iff Q_m(e, \bar{x}, (y)_0, (y)_1)$$
$$U(y) = (y)_0 .$$

Then

$$f(\bar{x}) \simeq U(\mu y \ T_m(e, \bar{x}, y)) .$$

□

2.19 Consequences of the Kleene Normal Form Theorem

The fact that every partial Turing computable function can be written in the normal form of Theorem ??? has a number of important consequences.

Theorem 2.19.1. *A function is partial Turing computable iff it is partial recursive.*

Proof. We already know that every partial recursive function is Turing computable. For the converse, suppose f is an m-ary partial Turing computable function. Then for some e,

$$\begin{aligned} f(\bar{x}) &\simeq U(\mu y\, T_m(e,\bar{x},y)) \\ &\simeq U(\mu y\, [\overline{\mathrm{sg}}(C_{T_m}(e,\bar{x},y)) = 0]) \ . \end{aligned}$$

Then f is defined by composition from U and $\mu y\, [\overline{\mathrm{sg}}(C_{T_m}(e,\bar{x},y)) = 0]$. Further, $\mu y\, [\overline{\mathrm{sg}}(C_{T_m}(e,\bar{x},y)) = 0]$ is defined by unbounded search from the primitive recursive function $\overline{\mathrm{sg}}(C_{T_m}(e,\bar{x},y))$. □ □

Definition 2.19.2. For $m, e \in \mathbb{N}$, ϕ_e^m is the m-ary partial recursive function defined by:

$$\phi_e^m(\bar{x}) \simeq U(\mu y\, T_m(e,\bar{x},y)) \ .$$

Remark 2.19.3. If e is not the Gödel number of a Turing Machine, then $\operatorname{dom} \phi_e^m = \emptyset$.

Theorem 2.19.4 (Kleene Enumeration Theorem)**.** *For each $m \in \mathbb{N}$,*

$$\phi_0^m, \phi_1^m, \ldots, \phi_n^m, \ldots$$

is a listing (with repetitions) of all partial recursive m-ary functions.

Proof. Each ϕ_e^m is a partial recursive function since it is defined by unbounded search and composition from the primitive recursive relation T_m. On the other hand, if f is an m-ary partial recursive function, then by the Kleene Normal Form Theorem, there is some e such that for all $\bar{x}$,

$$\begin{aligned} f(\bar{x}) &\simeq U(\mu y\, T_m(e,\bar{x},y) \\ &\simeq \phi_e^m(\bar{x}) \ . \end{aligned}$$

□

Theorem 2.19.5 (Universal Turing Machine)**.** *There is a Turing Machine T such that if:*

f is a 1-ary partial Turing computable function,

M is a Turing Machine that computes f,

e is the Gödel number of M,

then T on initial configuration

$$^{\infty}001^{e+1}01^{x+1}00^{\infty}$$

has

a) terminal configuration

$$^{\infty}001^{e+1}01^{x+1}01^{f(x)+1}00^{\infty}$$

if $f(x)$ is defined, and

b) has no computation if $f(x)$ is not defined.

Proof. Let

$$\begin{aligned} g(e,x) &\simeq \phi^1_e(x) \\ &\simeq U(\mu y\, T_1(e,x,y)) \ . \end{aligned}$$

Then g is a partial recursive 2-ary function. Let T be a Turing Machine that computes g. Now if f is a partial recursive 1-ary function and M is a Turing Machine that computes f, then T satisfies (a) and (b). □

Theorem 2.19.6. *Let R be an n-ary relation on $\mathbb{N}$. The following are equivalent:*

a) R is r.e.;

b) for some primitive recursive P,

$$R(\bar{x}) \iff \exists y\, P(\bar{x},y) \ ;$$

c) for some recursive P,

$$R(\bar{x}) \iff \exists y\, P(\bar{x},y) \ ;$$

Proof. (a)⇒(b): Let R be r.e. Then $R = \mathrm{dom}(f)$ for some partial recursive f. For some e,

$$f(\bar{x}) \simeq U(\mu y\, T_m(e, \bar{x}, y)) \text{ , so}$$

$$\begin{aligned} R(\bar{x}) &\iff f(\bar{x}) \text{ is defined} \\ &\iff \exists y\, T_m(e, \bar{x}, y) \ . \end{aligned}$$

(b)⇒(c): Every primitive recursive relation is recursive. (c)⇒(a): Let P be a recursive relation such that

$$R(\bar{x}) \iff \exists y\, P(\bar{x}, y) \ .$$

Let $f(\bar{x}) \simeq \mu\, y\, P(\bar{x}, y)$. Then

$$\begin{aligned} f(\bar{x}) \text{ is defined} &\iff \exists y\, P(\bar{x}, y) \\ &\iff R(\bar{x}) \ . \end{aligned}$$

□

Exercise 2.19.7 (Selection)**.** Let R be an r.e. 2-ary relation. Show that there is a partial recursive 1-ary function f such that:

a) $f(x)$ is defined iff $\exists y\, R(x, y)$, and

b) if $\exists y\, R(x, y)$ then $R(x, f(x))$.

Note. If we define f by $f(x) \simeq \mu y\, R(x, y)$ then f satisfies (a) and (b), but f might not be partial recursive.

Exercise 2.19.8. Suppose f is partial recursive. Then there is a finite sequence

$$f_0, f_1, \dots, f_n$$

such that for each $i \leq n$, f_i is either an initial function, or is obtained from previous functionsin the sequence by composition, primitive recursion, or unbounded search. Explain why it is possible to choose $f_0, \dots, f_n$ so that unbounded search appears at most once.

Theorem 2.19.9. *Let R be a relation on $\mathbb{N}$. Then R is recursive iff both R and $\neg R$ are r.e.*

Proof. ($\Rightarrow$) : Suppose R be recursive. Let

$$f(\bar{x}) \simeq \mu\, y\, [\overline{sg}(C_R(\bar{x})) + y = 0]\ .$$

Then f is partial recursive and $\mathrm{dom}(f) = \neg R$. Hence $\neg R$ is r.e. Similarly, R is recursive, and so R is r.e. ($\Leftarrow$) : Suppose both R and $\neg$ are r.e. Then there are recursive relations P and Q such that:

$$R(\bar{x}) \iff \exists y\, P(\bar{x}, y)\ ; \tag{2.192}$$

$$\neg R(\bar{x}) \iff \exists y\, Q(\bar{x}, y)\ . \tag{2.193}$$

Now,

$\forall \bar{x}\ [R(\bar{x}) \vee \neg R(\bar{x})$; hence

$\forall \bar{x}\ [\exists y\, P(\bar{x}, y) \vee \exists y\, Q(\bar{x}, y)]$; so,

$\forall \bar{x}\, \exists y\ [P(\bar{x}, y) \vee Q(\bar{x}, y)]$.

Let

$$f(\bar{x}) \simeq \mu y\, [P(\bar{x}, y) \vee Q(\bar{x}, y)]\ .$$

Then f is recursive.

Claim. $R(\bar{x}) \iff P(\bar{x}, f(\bar{x}))$ (and hence R is recursive).

Proof of Claim ($\Rightarrow$): Suppose $R(\bar{x})$; then

$$P(\bar{x}, f(\bar{x})) \vee Q(\bar{x}, f(\bar{x}))\ .$$

If $\neg P(\bar{x}, f(\bar{x}))$, then $Q(\bar{x}, f(\bar{x}))$; hence $\exists y\, Q(\bar{x}, y)$. But then $\neg R(\bar{x})$, which is a contradiction. □Claim

Hence, $P(\bar{x}, f(\bar{x}))$.

($\Leftarrow$) : Suppose $P(\bar{x}, f(\bar{x}))$; then $\exists y\, P(\bar{x}, y)$; hence $R(\bar{x})$. □

Proposition 2.19.10. *For every relation R on $\mathbb{N}$,*

a) $\exists y\, \exists z\, R(\bar{x}, y, z) \iff \exists u\, R(\bar{x}, (u)_0, (u)_1)$*;*

b) $\forall y\, \forall z\, R(\bar{x}, y, z) \iff \forall u\, R(\bar{x}, (u)_0, (u)_1)$.

Note that if $R(\bar{x}, y, z)$ is recursive (respectively primitive recursive) then so is $R(\bar{x}, (u)_0, (u)_1)$. Consequently:

Theorem 2.19.11 ($\exists$-closure of r.e. sets)**.** *If $R(\bar{x}, y)$ is r.e., then so is $\exists y\, R(\bar{x}, y)$.*

Proof. Let R be r.e. Then there is a recursive P such that:

$$R(\bar{x}, y) \iff \exists w\, P(\bar{x}, y, w)\ .$$

Hence,

$$\exists y\, R(\bar{x}, y) \iff \exists y\, \exists w\, P(\bar{x}, y, w)\ .$$

By 2.9, R is r.e. □

Proposition 2.19.12. *For every relation R,*

a) $\forall y < z\, \exists w\, R(\bar{x}, y, z, w) \iff \exists w\, \forall y < z\, R(\bar{x}, y, z, (w)_y)$*;*

b) $\exists y < z\, \forall w\, R(\bar{x}, y, z, w) \iff \forall w\, \exists y < z\, R(\bar{x}, y, z, (w)_y)$

Proposition 2.19.13 (closure of r.e. sets under bounded quantification)**.**

a) If $R(\bar{x}, y, z)$ is recursive then so are $\forall y < z\, R(\bar{x}, y, z)$ and $\exists y < z\, R(\bar{x}, y, z)$.

b) If $R(\bar{x}, y, z)$ is r.e. then so are $\forall y < z\, R(\bar{x}, y, z)$ and $\exists y < z\, R(\bar{x}, y, z)$.

Proof. The proof of (a) is the same as the corresponding proof for primitive recursive relations R. To prove (b), suppose R is r.e. Then for some primitive recursive P, using ??? we have:

$$\begin{aligned} \forall y < z\, R(\bar{x}, y, z) &\iff \forall y < z\, \exists w\, P(\bar{x}, y, z, w) && (2.194)\\ &\iff \exists w\, \forall y < z\, P(\bar{x}, y, z, (w)_y)\ . && (2.195) \end{aligned}$$

□

Exercise 2.19.14. Let f be partial recursive. Show that the range of f is r.e.

Exercise 2.19.15. Let f be a strictly increasing recursive 1-ary function. Show that the range of f is recursive.

Theorem 2.19.16. *Let X be a nonempty subset of $\mathbb{N}$. The following are equivalent:*

a) X is r.e.;

b) X is the range of a primitive recursive 1-ary function;

c) X is the range of a recursive 1-ary function;

d) X is the range of a partial recursive 1-ary function.

Proof. $(a) \Rightarrow (b)$: If X is r.e. then for some primitive recursive P,

$$n \in X \iff \exists x\, P(n, x)\ .$$

Let $a_0 \in X$, and define f by:

$$f(x) = \begin{cases} (x)_0, & \text{if } P((x)_0, (x)_1); \\ a_0, & \text{otherwise.} \end{cases}$$

Then f is primitive recursive, and $\operatorname{ran}(f) = X$.
$((b) \Rightarrow (c) \Rightarrow (d))$ is trivial.
$(d) \Rightarrow (a)$: This is Exercise ??? □

Definition 2.19.17. For $e \in \mathbb{N}$,

$$\begin{aligned} W_e &= \{x : \exists y\, T_1(e, x, y)\} \\ &= \operatorname{dom} \phi^1_e\ . \end{aligned}$$

Proposition 2.19.18. *$W_0, W_1, \ldots, W_n, \ldots$ is an enumeration (with repetitions) of all r.e. subsets of $\mathbb{N}$.*

The following is a formal version of Exercise ???.

Theorem 2.19.19. *There exists subsets of $\mathbb{N}$ that are r.e. but not recursive.*

Proof. Let

$$C = \{n : n \in W_n\}\ .$$

Then C is r.e. Suppose C is recursive. Then $\mathbb{N} \setminus C$ is r.e. So for some e, and all x,

$$\begin{aligned} x \notin C &\iff x \in \mathbb{N} \setminus C \\ &\iff x \in W_e\ . \end{aligned}$$

But then:

$$\begin{aligned} e \notin C &\iff e \in \mathbb{N} \setminus C \\ &\iff e \in W_e\ . \end{aligned}$$

But this gives the contradiction:

$$\begin{aligned} e \notin C &\iff e \in W_e \\ &\iff e \in C \,. \qquad \square \end{aligned}$$

□

Remark 2.19.20. Assuming Church's Thesis, the existence of a r.e. set that is not recursive is equivalent to the existence of an effectively enumerable set that is not effectively decidable.

Appendix A

Sentential Connectives

Definition A.0.1. Suppose P and Q are sentences. From P and Q we can form new compound sentences using *and*, *or*, *not*, etc. These words are called *sentential connectives* because they 'connect' sentences together, forming new sentences. It is convenient to use the following abbreviations:

$P \mathbin{\&} Q$ is an abbreviation of: *P and Q*;

P or Q is an abbreviation of: *P or Q*;

$\neg P$ is an abbreviation of: *not P*;

$P \implies Q$ is an abbreviation of: *if P then Q*;

$P \iff Q$ is an abbreviation of: *P if and only if Q*;[1]

1. $P \mathbin{\&} Q$ is the *conjunction* of P and Q;
2. P or Q is the *disjunction* of P and Q;
3. $\neg P$ is the *negation* of P.

How do we decide whether these compound statements are true or false? The truth or falsity of these compound statements is completely determined by the truth or falsity of the components P and Q.

1. $P \mathbin{\&} Q$ is true iff both P and Q are true.

[1] *if and only if* is often also abbreviated by *iff*.

2. P *or* Q is true iff P is true or Q is true.[2]
3. $\neg P$ is true iff P is false.
4. $P \implies Q$ is true iff the truth of P implies the truth of Q.[3]

The rest of this section is pictures of some truth tables that I have left out

[2] P *or* Q is true if one or *both* of P and Q are true.

[3] So to show that $P \implies Q$ is true, we usually *assume* that P is true, and then show that Q is true. What happens if P is false? In this case we say that $P \implies Q$ is true. (So the only way that $P \implies Q$ can be false is if P is true and Q is false.)

Appendix B

Existential and Universal Quantifiers

Definition B.0.1. It is often convenient to use the following abbreviations:

$\forall x \dots$	is an abbreviation of	*for every* $x \dots$[1]
$\forall x \in A \dots$	is an abbreviation of	*for every* $x \in A \dots$
$\exists x \dots$	is an abbreviation of	*there exists an* x *such that* $\dots$[2]
$\exists x \in A \dots$	is an abbreviation of	*there exists an* $x \in A$ such that $\dots$

a) $\forall$ is called the *universal quantifier* symbol;

b) $\exists$ is called the *existential quantifier* symbol.

[1] The following have the same meaning:

a) *for every* $x \dots$

b) *for each* $x \dots$

c) *for all* $x \dots$.

[2] The following have the same meaning:

a) *there exists an* x such that $\dots$

b) *there is an* x such that $\dots$

c) *there is some* x such that $\dots$

d) *for some* $x \dots$.

The statement

1. *For every real number x there is a real number y such that $x < y$*

can be abbreviated by

1. $\forall x \in \mathbb{R}\, \exists y \in \mathbb{R}\ [x < y]$.

We shall sometimes use the notation $P(x)$ (or $Q(x)$ or $R(x)$) to stand for a mathematical statement that talks about x. For example, $P(x)$ might be the statement: $x^2 < 3$. Then

$$\forall x \in \mathbb{R}\ P(x)$$

is the statement

For every real number x, $x^2 < 3$,

or equivalently,

The square of every real number is less than 3 .

This is clearly a false statement.[3] Similarly, we shall sometimes use the notation $P(x, y)$ (or $Q(x, y)$ or $R(x, y)$) to stand for a mathematical statement about both x and y. For example, $Q(x, y)$ might stand for the statement: $x < y$.[4] There is a

[3]In ordinary English the following two sentences have the same meaning:

1. For every $x \in \mathbb{R}$, $P(x)$
2. $P(x)$ for every $x \in \mathbb{R}$.

However, when we use abbreviations we always write this as

$$\forall x \in \mathbb{R}\ P(x).$$

We do *not* write it as

$$P(x)\ \forall x \in \mathbb{R}.$$

The reason is that in a more complicated statement, if we were to write

$$Q(x)\ \forall x \in \mathbb{R}\ R(x)$$

it would be unclear whether the $\forall$ was talking about Q or R. Similarly, $\exists$ is always placed in front of the statement to which it refers.

[4]Mathematical statements can talk about more than two objects x and y. For example, suppose that the following statement is true.

$$\forall x \in A\, \forall y \in A\, \forall z \in A\ [z = x \textit{ or } z = y]\ .$$

What does this tell you about the set A?

close connection between the meanings of $\forall$ and $\exists$:

Remark B.0.2.

a) $\neg\ [\forall x\ P(x)]$ is true iff $\exists x\ \neg P(x)$ is true;

b) $\neg\ [\exists x\ P(x)]$ is true iff $\forall x\ \neg P(x)$ is true. Similarly,

c) $\neg\ [\forall x \in A\ P(x)]$ is true iff $\exists x \in A\ \neg P(x)$ is true;

d) $\neg\ [\exists x \in A\ P(x)]$ is true iff $\forall x \in A\ \neg P(x)$ is true.

Proof. $(a) \Longrightarrow)$: Suppose the statement

$$\neg\ [\forall x\ P(x)]$$

is true. Then

$$\forall x\ P(x)$$

is false. This means that it is false that

$$\text{for } \textit{every } x,\ [P(x) \text{ is true}]$$

But then there must be *some* x such that $P(x)$ is false, which is in turn equivalent to saying that

$$\exists x\ \neg P(x)$$

is true. The proof in the direction $(\Longleftarrow)$ is similar. You should write out for yourself the reasoning for (b), (c), and (d). □

We can combine ??? and ??? to simplify the negation of complicated sentences. For example, using repeatedly, we see that each of the following are equivalent:

$$\neg\big[\forall x \in A\, \exists y \in B\, \forall z \in C\ [P(x,y) \Longrightarrow Q(z)]\big]$$
$$\exists x \in A\, \neg\big[\exists y \in B\, \forall z \in C\ [P(x,y) \Longrightarrow Q(z)]\big]$$
$$\exists x \in A\, \forall y \in B\, \neg\big[\forall z \in C\ [P(x,y) \Longrightarrow Q(z)]\big]$$
$$\exists x \in A\, \forall y \in B\, \exists z \in C\, \neg[P(x,y) \Longrightarrow Q(z)]$$
$$\exists x \in A\, \forall y \in B\, \exists z \in C\ [P(x,y)\ \&\ \neg\, Q(z)]$$

Note how each $\forall$ has changed to $\exists$, how each $\exists$ has changed to $\forall$, and how $\neg$ has moved 'inside.'

Appendix C

Mathematical Induction

C.1 How to show that infinitely many math statements are true

Often we are dealing with infinitely many mathematical statements at the same time. For example, consider the following infinite list of statements:

$$(0+1)^2 = 0^2 + 2 \times 0 + 1 \tag{0}$$

$$(1+1)^2 = 1^2 + 2 \times 1 + 1 \tag{1}$$

$$(2+1)^2 = 2^2 + 2 \times 2 + 1 \tag{2}$$

$$(3+1)^2 = 3^2 + 2 \times 3 + 1 \tag{3}$$

etc., where the *n*-th statement is

$$(n+1)^2 = n^2 + 2n + 1\,. \tag{n}$$

(We call here the statement $(0+1)^2 = 0^2 + 2 \times 0 + 1$ the 0-th statement.) This infinite list of statements is equivalent to the *single* statement:

$$\forall n \in \mathbb{N}\ [(n+1)^2 = n^2 + 2n + 1]\,.$$

To show that this statement is true we reason as follows: Given any $n \in \mathbb{N}$, we want to show that $(n+1)^2 = n^2 + 2n + 1$. Using the rules for multiplication and addition of numbers we 'multiply out' the left hand side of the equation and see that it is equal to the right side. *Since the integer n that we were given was arbitrary* (that is we made no special assumptions about n other than that it belonged to $\mathbb{N}$), we conclude that the equation is true for *every* integer n. We have just learned one method of proving statements of the form

$$\forall n \in \mathbb{N}\ P(n)\,.$$

To do this, we suppose that we are given an arbitrary $n \in \mathbb{N}$, and show that $P(n)$ is true. However, this method often does not work, since in many cases we are unable to show directly that $P(n)$ is true. The following principle of *mathematical induction* is a very powerful technique for handling such cases.

Example C.1.1. Consider the following statement: For every $n \in \mathbb{N}$,

$$1 + \cdots + n = \frac{n(n+1)}{2}\,.$$

This statement is equivalent to the infinite list of statements:

$$0 = \frac{0 \times 1}{2} \tag{0}$$

$$1 = \frac{1 \times 2}{2} \tag{1}$$

$$1 + 2 = \frac{2 \times 3}{2} \tag{2}$$

$$1 + 2 + 3 = \frac{3 \times 4}{2} \tag{3}$$

$$1 + 2 + 3 + 4 = \frac{4 \times 5}{2} \tag{4}$$

etc., where the n-th statement is

$$1 + \cdots + n = \frac{n(n+1)}{2}\,.$$

Now you can check each of these statements (0),(1),(2),(3), and (4), and see that they are true. And you can check statements (5), (6), and (234) if you want to (even though they have not been written down here). But no matter how much time you spend, you can never (by checking them one at a time), check *all* of them. What is needed is a method that will show that each of the infinitely many statements is true, without having to check each statement individually. Let $P(0)$ be the statement that is the equation of line (0); that is, $P(0)$ stands for the equation $0 = \frac{0\times 1}{2}$. Similarly, let $P(1)$ be the equation of line (1). More generally, let $P(n)$ be the equation of line (n). We want to show that for every integer n, $P(n)$ is true. Suppose we are able to show the following:

1. a) $P(0)$ is true;[1]
2. b) for every $n \in \mathbb{N}$, $P(n) \Longrightarrow P(n+1)$ is true.

(b) is a statement about *all* integers that is equivalent to the following infinite list of statements:

1. if $P(0)$ is true then $P(1)$ is true; $(b.0)$
2. if $P(1)$ is true then $P(2)$ is true; $(b.1)$
3. if $P(2)$ is true then $P(3)$ is true; $(b.2)$
4. etc.

Now (a) says that $P(0)$ is true. But then by $(b.0)$, $P(1)$ must also be true. And if $P(1)$ is true, then by $(b.1)$, $P(2)$ must be true. Continuing, we see that for every non-negative integer n, $P(n)$ is true. And this is what we wanted to show.
In order to complete the argument, we still need to show (b). This can be done by using the method of the previous section. To show (b), we reason as follows: In order to show that for every $n \in \mathbb{N}$, $P(n) \Longrightarrow P(n+1)$ is true, we suppose that an arbitrary integer n is given. Then, assume that $P(n)$ is true. If we can show that $P(n+1)$ is true, we will have shown that statement b) is true.
So suppose we are given n. Assume that $P(n)$ is true, that is, assume

$$1 + \cdots + n = \frac{n(n+1)}{2}\,.$$

We want to show that $P(n+1)$ is true, that is we want to show

$$\underbrace{1 + \cdots + n} + (n+1) = \frac{(n+1)(n+2)}{2}\,.(*)$$

[1]We actually already know this.

Since $P(n)$ is true, we can replace $\underbrace{1+\cdots+n}$ in $(*)$ by $\frac{n(n+1)}{2}$. The left side of $(*)$ now becomes

$$\frac{n(n+1)}{2} + (n+1)\,.$$

A little bit of arithmetic now shows that the left side is equal to the right side of $(*)$.[2]

We can use the same reasoning to formulate a general principle as follows:

Principle of Mathematical Induction. Let $P(n)$ be a mathematical statement. If

a) $P(0)$ is true, and

b) for every integer n,
$$P(n) \Longrightarrow P(n+1) \textit{ is true,}$$

then $P(n)$ is true for every $n \in \mathbb{N}$.

Remark C.1.2. A proof of a mathematical statement using the Principle of Mathematical Induction is called a *proof by mathematical induction*, or simply a *proof by induction*.

Remark C.1.3. A proof by induction consists of two parts: the *Basis*, and the *Induction Step*.

1. *Basis:* Show that the statement $P(0)$ is true.
2. *Induction Step:* Given $n \in \mathbb{N}$,
   a) Assume that $P(n)$ is true.[3] *(induction hypothesis)*
   b) Show that $P(n+1)$ is true.

Such a proof is often called a *proof by induction on n*.

[2]The fact that for every $n \in \mathbb{N}$, $1+\cdots+n = \frac{n(n+1)}{2}$, was used in Calc. II, where a different proof was given. Do you remember how this fact was used?

[3]The assumption that $P(n)$ is true is called the *induction hypothesis.*

Example C.1.4. We show that $7^n - 4^n$ is a multiple of 3 for every non-negative integer n. Let $P(n)$ be the statement

$$7^n - 4^n \textit{ is a multiple of } 3\,.$$

We give a proof by induction on n.

1 *Basis:* $P(0)$ is the statement

$$7^0 - 4^0 \textit{ is a multiple of } 3\,,$$

which is a true statement.

2 *Induction Step:* Given $n \in \mathbb{N}$,

a) assume $P(n)$ is true. This is the induction hypothesis. That is *assume*

$$7^n - 4^n \textit{ is a multiple of } 3\,.$$

b) We want to show that $P(n+1)$ is true, that is, we want to show

$$7^{n+1} - 4^{n+1} \textit{ is a multiple of } 3\,. \tag{$*$}$$

We have

$$\begin{aligned} 7^{n+1} - 4^{n+1} &= \left(7^{n+1} - 7 \cdot 4^n\right) + \left(7 \cdot 4^n - 4 \cdot 4^n\right) && \text{(C.1)} \\ &= 7(\underbrace{7^n - 4^n}_{(a)}) + 4^n(\underbrace{7-4}_{(b)})\,. && \text{(C.2)} \end{aligned}$$

By the induction hypothesis, (a) is divisible by 3, and clearly (b) is divisible by 3. Therefore the left side of $(C.1)$ is divisible by 3, which shows $(*)$.

C.2 Other versions of mathematical induction

Suppose we wish to prove that a certain statement $P(n)$ is true for every positive integer greater than or equal to 3. This can be done if we can show that $P(3)$ is true, and that whenever $P(n)$ is true, then $P(n+1)$ is true. More generally, we have the following version of mathematical induction:

Theorem C.2.1. *Let $k \in \mathbb{Z}$. If*

a) $P(k)$ is true, and

b) for every integer $n \geq k$,

$$P(n) \Longrightarrow P(n+1) \text{ is true,}$$

then $P(n)$ is true for every integer $n \geq k$.

Remark C.2.2. A proof that uses this version of mathematical induction is often called a *proof by induction on n, for $n \geq k$.*

The following is another version of mathematical induction that is actually equivalent to the original version.

Theorem C.2.3 (Strong Induction)**.** *If*

a) $P(0)$ is true, and

b) for each $n \in \mathbb{N}$, if $\big[P(i)$ is true for every i such that $1 \leq i \leq n\,\big]$, then $P(n+1)$ is true,

then $P(n)$ is true for every $n \in \mathbb{N}$.

The following is an important example of a proof by strong induction.

Theorem C.2.4 (The Well-Ordering Property of $\mathbb{N}$))**.** *Every nonempty subset of $\mathbb{N}$ has a smallest member.*[4]

Proof. Let $\emptyset \neq A$ be a subset of $\mathbb{N}$. *Suppose A has no smallest member.* $(*)$ We obtain a contradiction by showing that $A = \emptyset$. From the contradiction it follows that the supposition 'A has no smallest member' must be false. But that means A must have a smallest member. So we assume $(*)$, and want to show that $A = \emptyset$. Let $P(n)$ be the statement: $n \notin A$. We show by strong induction on n that $P(n)$ is true for all $n \in \mathbb{N}$.

[4] b is the smallest member of a subset A of $\mathbb{N}$ if:

1) $b \in A$, and

2) for every $n \in A$, $b \leq n$.

1. *Basis:* $P(0)$ is true, i.e. $0 \notin A$.[5]

2. *Induction Step:* Given $n \in \mathbb{N}$, assume that $P(i)$ is true for all i such that $1 \leq i \leq n$, i.e. assume

   $$i \notin A \textit{ for all } i \textit{ such that } 1 \leq i \leq n. \qquad \textit{(induction hypothesis)}$$

   Then $n+1 \notin A$, that is, $P(n+1)$ is true.[6] By Strong Induction, $P(n)$ is true for every $n \in \mathbb{N}$, i.e. $n \notin A$ for every $n \in \mathbb{N}$, i.e. $A = \emptyset$.

□

Remark C.2.5. Notice that in the above proof we assumed that the statement we were trying to prove (i.e. the statement 'A has a smallest member'), was false! From the assumption that the statement was false, we arrived at a contradiction. We then concluded that the statement must be true. Such reasoning is called *a proof by contradiction.*

[5]If $0 \in A$, then 0, being the smallest member of $\mathbb{N}$, would also be the smallest member of A.
[6]If $n+1 \in A$ and $i \notin A$ for all $1 \leq i \leq n$, then $n+1$ would be the smallest member of A.