Chapter 1
Mathematical Preliminaries

In mathematics you don't understand things. You just get used to them.

– J. von Neumann

Abstract The goal of this chapter is to give a fairly comprehensive overview of the theory that underpins the rest of the book. On first reading, it may seem a little dry and is often excluded in other similar books. However, without a solid understanding of logic and representation of numbers it seems clear that constructing digital circuits to put this theory into practise would be much harder. The theory here will present an introduction to propositional logic, sets and functions, number systems and Boolean algebra. These four main areas combine to produce a basis for formal methods to describe, manipulate and implement digital systems such as computer processors. Those with a background in mathematics or computer science might skip this material and use it simply for reference; those approaching the subject from another background would be advised to read the material in more detail.

1.1 Propositions and Predicates

Definition 1. A **proposition** is a statement whose meaning, termed the **truth value**, is either true or false. Less formally, we say the statement is true if it has a truth value of true and false if it has a truth value of false.

A **predicate** is a proposition which contains one or more **variables**; only when concrete values are **assigned** to each of the variables can the predicate be called a proposition.

Since we use them so naturally, it almost seems too formal to define what a proposition is. However, by doing so we can start to use them as a building block to describe what logic is and how it works. The statement

"the temperature is $90°C$"

is a proposition since it is definitely either true or false. When we take a proposition and decide whether it is true or false, we say we have **evaluated** it. However, there are clearly a lot of statements that are not propositions because they do not state any proposal. For example,

“turn off the heat”

is a command or request of some kind, it does not evaluate to a truth value. Propositions must also be well defined in the sense that they are definitely either true or false, i.e., there are no “gray areas” in between. The statement

“$90^\circ C$ is too hot”

is not a proposition since it could be true or false depending on the context, or your point of view: $90^\circ C$ probably is too hot for body temperature but probably not for a cup of coffee. Finally, some statements look like propositions but cannot be evaluated because they are paradoxical. The most famous example of this situation is the liar paradox, usually attributed to the Greek philosopher Eubulides, who stated it as

“a man says that he is lying, is what he says true or false ?”

although a clearer version is the more commonly referenced

“this statement is false” .

If the man is telling the truth, everything he says must be true which means he is lying and hence everything he says is false. Conversely, if the man is lying everything he says is false, so he cannot be lying since he said he was ! In terms of the statement, we cannot be sure of the truth value so this is not normally classed as a proposition.

As stated above, a predicate is just a proposition that contains variables. By **assigning** the variable a value we can turn the predicate into a proposition and evaluate the corresponding truth value. For example, consider the predicate

“$x^\circ C$ equals $90^\circ C$”

where x is a variable. By assigning x a value we get a proposition; setting $x = 10$, for example, gives

“$10^\circ C$ equals $90^\circ C$”

which clearly evaluates to false. Setting $x = 90^\circ C$ gives

“$90^\circ C$ equals $90^\circ C$”

which evaluates to true. In some sense, a predicate is an abstract or general proposition which is not well defined until we assign values to all the variables.

1.1.1 Connectives

Definition 2. A **connective** is a statement which binds single propositions into a compound proposition. For brevity, we use symbols to denote common connectives:

- "not x" is denoted $\neg x$.
- "x and y" is denoted $x \wedge y$.
- "x or y" is denoted $x \vee y$, this is usually called an inclusive-or.
- "x or y but not x and y" is denoted $x \oplus y$, this is usually called an exclusive-or.
- "x implies y" is denoted $x \rightarrow y$, which is sometimes written as "if x then y".
- "x is equivalent to y" is denoted $x \leftrightarrow y$, which is sometimes written as "x if and only if y" or further shortened to "x iff. y".

Note that we group statements using parentheses when there could be some confusion about the order they are applied; hence $(x \wedge y)$ is the same as $x \wedge y$.

A proposition or predicate involving connectives is built from **terms**; the connective joins together these terms into an **expression**. For example, the expression

"the temperature is less than 90°C $\wedge$ the temperature is greater than 10°C"

contains two terms that propose

"the temperature is less than 90°C"

and

"the temperature is greater than 10°C" .

These terms are joined together using the $\wedge$ connective so that the whole expression evaluates to true if both of the terms are true, otherwise it evaluates to false. In a similar way we might write a compound predicate

"the temperature is less than x°C $\wedge$ the temperature is greater than y°C"

which can only be evaluated when we assign values to the variables x and y.

Definition 3. The meaning of connectives is usually describe in a tabular form which enumerates the possible values each term can take and what the resulting truth value is; we call this a **truth table**.

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \rightarrow y$	$x \leftrightarrow y$
false	false	true	false	false	false	true	true
false	true	true	false	true	true	true	false
true	false	false	false	true	true	false	false
true	true	false	true	true	false	true	true

The $\neg$ connective negates the truth value of an expression so considering

$$\neg(x > 10)$$

we find that the expression $\neg(x > 10)$ is true if the term $x > 10$ is false and the expression is false if $x > 10$ is true. If we assign $x = 9$, $x > 10$ is false and hence the expression $\neg(x > 10)$ is true. If we assign $x = 91$, $x > 10$ is true and hence the expression $\neg(x > 10)$ is false.

The meaning of the $\wedge$ connective is also as one would expect; the expression

$$(x > 10) \wedge (x < 90)$$

is true if *both* the expressions $x > 10$ and $x < 90$ are true, otherwise it is false. So if $x = 20$, the expression is true. But if $x = 9$ or $x = 91$, then it is false: even though one or other of the terms is true, they are not both true.

The inclusive-or and exclusive-or connectives are fairly similar. The expression

$$(x > 10) \vee (x < 90)$$

is true if *either* $x > 10$ *or* $x < 90$ is true or both of them are true. Here we find that all the assignments $x = 20$, $x = 9$ and $x = 91$ mean the expression is true; in fact it is hard to find an x for which it evaluates to false ! Conversely, the expression

$$(x > 10) \oplus (x < 90)$$

is only true if *only one* of either $x > 10$ or $x < 90$ is true; if they are both true then the expression is false. We now find that setting $x = 20$ means the expression is false while both $x = 9$ and $x = 91$ mean it is true.

Inference is a bit more tricky. If we write $x \rightarrow y$, we usually call x the **hypothesis** and y the **conclusion**. To justify the truth table for inference in words, consider the example

$$(x \text{ is prime }) \wedge (x \neq 2) \rightarrow (x \equiv 1 \pmod 2)$$

i.e., if x is a prime other than 2, it follows that it is odd. Therefore, if x is prime then the expression is true if $x \equiv 1 \pmod 2$ and false otherwise since the implication is invalid. If x is not prime, the expression does not really say anything about the expected outcome: we only know what to expect if x was prime. Since it could still be that $x \equiv 1 \pmod 2$ even when x is not prime, and we do not know anything better from the expression, we assume it is true when this case occurs.

Equivalence is fairly simple. The expression $x \leftrightarrow y$ is only true if x and y evaluate to the same value. This matches the idea of equality of numbers. As an example, consider

$$(x \text{ is odd }) \leftrightarrow (x \equiv 1 \pmod 2).$$

This expression is true since if the left side is true, the right side must also be true and vice versa. If we change it to

$$(x \text{ is odd }) \leftrightarrow (x \text{ is prime }),$$

then the expression is false. To see this, note that only some odd numbers are prime: just because a number is odd does not mean it is always prime although if it is prime it must be odd (apart from the corner case of $x = 2$). So the equivalence works in one direction but not the other and hence the expression is false.

Definition 4. We call two expressions logically **equivalent** if they are composed of the same variables and have the same truth value for every possible assignment to those variables.

An expression which is equivalent to true, no matter what values are assigned to any variables, is called a **tautology**; an expression which is equivalent to false is called a **contradiction**.

Some subtleties emerge when trying to prove two expressions are logically equivalent. However, we will skirt around these. For our purposes it suffices to simply enumerate all possible values each variable can take, and check the two expressions produce identical truth values in all cases. In practise this can be hard since with n variables there will be 2^n possible assignments, an amount which grows quickly as n grows ! More formally, two expressions x and y are only equivalent if $x \leftrightarrow y$ can be proved a tautology.

1.1.2 Quantifiers

Definition 5. A **free variable** in a given expression is one which has not yet been assigned a value. Roughly speaking, a **quantifier** is a statement which allows a free variable to take one of many values:

- the **universal quantifier** "for all x, y is true" is denoted $\forall\, x\ [y]$.
- the **existential quantifier** "there exists an x such that y is true" is denoted $\exists\, x\ [y]$.

We say that applying a quantifier to a variable quantifies it; after it has been quantified we say it has been **bound**.

Put more simply, when we encounter an expression such as

$$\exists\, x\ [y]$$

we are essentially assigning x all possible values; to make the expression true *just one* of these values needs to make the expression y true. Likewise, when we encounter

$$\forall\, x\ [y]$$

we are again assigning x all possible values. This time however, to make the expression true, *all of them* need to make the expression y true. Consider the following example:

$$\text{"there exists an } x \text{ such that } x \equiv 0 \pmod 2\text{"}$$

which we can re-write symbolically as

$$\exists\, x\, [x \equiv 0 \pmod 2].$$

In this case, x is bound by the $\exists$ quantifier; we are asserting that for some value of x it is true that $x \equiv 0 \pmod 2$. To make the expression true just one of these values needs to make the term $x \equiv 0 \pmod 2$ true. The assignment $x = 2$ satisfies this so the expression is true. As another example, consider the expression

"for all x, $x \equiv 0 \pmod 2$"

which we re-write

$$\forall\, x\, [x \equiv 0 \pmod 2].$$

Here we are making a more general assertion about x by saying that for all x, it is true that $x \equiv 0 \pmod 2$. To decide if this particular expression is false, we need simply to find an x such that $x \not\equiv 0 \pmod 2$. This is easy since any odd value of x is good enough. Therefore the expression is false.

Definition 6. Informally, a **predicate function** is just a shorthand way of writing predicates; we give the function a name and a list of free variables. So for example the function

$$f(x,y) : x = y$$

is called f and has two variables named x and y. If we use the function as $f(10,20)$, performing the binding $x = 10$ and $y = 20$, it has the same meaning as $10 = 20$.

1.1.3 Manipulation

Definition 7. Manipulation of expressions is governed by a number of axiomatic laws:

equivalence	$x \leftrightarrow y$	$\equiv (x \rightarrow y) \wedge (y \rightarrow x)$
implication	$x \rightarrow y$	$\equiv \neg x \vee y$
involution	$\neg\neg x$	$\equiv x$
idempotency	$x \wedge x$	$\equiv x$
commutativity	$x \wedge y$	$\equiv y \wedge x$
association	$(x \wedge y) \wedge z$	$\equiv x \wedge (y \wedge z)$
distribution	$x \wedge (y \vee z)$	$\equiv (x \wedge y) \vee (x \wedge z)$
de Morgan's	$\neg(x \wedge y)$	$\equiv \neg x \vee \neg y$
identity	$x \wedge \text{true}$	$\equiv x$
null	$x \wedge \text{false}$	$\equiv \text{false}$
inverse	$x \wedge \neg x$	$\equiv \text{false}$
absorption	$x \wedge (x \vee y)$	$\equiv x$
idempotency	$x \vee x$	$\equiv x$
commutativity	$x \vee y$	$\equiv y \vee x$
association	$(x \vee y) \vee z$	$\equiv x \vee (y \vee z)$
distribution	$x \vee (y \wedge z)$	$\equiv (x \vee y) \wedge (x \vee z)$
de Morgan's	$\neg(x \vee y)$	$\equiv \neg x \wedge \neg y$
identity	$x \vee \text{false}$	$\equiv x$
null	$x \vee \text{true}$	$\equiv \text{true}$
inverse	$x \vee \neg x$	$\equiv \text{true}$
absorption	$x \vee (x \wedge y)$	$\equiv x$

A common reason to manipulate a logic expression is to simplify it in some way so that it contains less terms or less connectives. A simplified expression might be more opaque, in the sense that it is not as clear what it means, but looking at things computationally it will generally be "cheaper" to evaluate. As a concrete example of simplification in action, consider the exclusive-or connective $x \oplus y$ which we can write as the more complicated expression

$$(y \wedge \neg x) \vee (x \wedge \neg y)$$

or even as

$$(x \vee y) \wedge \neg(x \wedge y).$$

The first alternative above uses five connectives while the second uses only four; we say that the second is simpler as a result. One can prove that these are equivalent by constructing truth tables for them. The question is, how did we get from one to the other by manipulating the expressions ?

To answer this, we simply start with one expression and apply our axiomatic laws to move toward the other. So starting with the first alternative, we try to apply the axioms until we get the second: think of the axioms like rules that allow one to re-write the expression in a different way. To start with, we can manipulate each term which looks like $p \wedge \neg q$ as follows

$$\begin{aligned} (p \wedge \neg q) &= (p \wedge \neg q) \vee \text{false} && \text{(identity)} \\ &= (p \wedge \neg q) \vee (p \wedge \neg p) && \text{(null + inverse)} \\ &= p \wedge (\neg p \vee \neg q) && \text{(distribution)} \end{aligned}$$

Using this new identity, we can re-write the whole expression as

$$\begin{aligned}(y \wedge \neg x) \vee (x \wedge \neg y) &= (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) \\ &= (x \vee y) \wedge (\neg x \vee \neg y) && \text{(collect)} \\ &= (x \vee y) \wedge \neg(x \wedge y) && \text{(de Morgan's)}\end{aligned}$$

which gives us the second alternative we are looking for. A later chapter shows how to construct an algorithm to do this sort of simplification mechanically so as to reduce our workload and reduce the chance of error.

1.2 Sets and Functions

The concept of a **set** and the theory behind such objects is a fundamental part of mathematics. Informally, a set is simply a well defined collection of **elements**. Here we mainly deal with sets of numbers, but it is important to note that the elements can be anything you want.

We can define a set using one of several methods. Firstly we can enumerate the elements, writing them down between a pair of braces. For example, one might define the set A of whole numbers between two and eight (inclusive) as

$$A = \{2, 3, 4, 5, 6, 7, 8\}.$$

The **cardinality** of a finite set is the number of elements it contains. For the set A, this is denoted by $|A|$ such that from the example above

$$|A| = 7.$$

If the element a is in the set A, we say a is a **member** of A or write

$$a \in A.$$

The ordering of the elements in the set does not matter, only their membership or non-membership. So we can define another set as

$$B = \{8, 7, 6, 5, 4, 3, 2\}$$

and be safe in the knowledge that $A = B$. However, note that elements cannot occur in a set more than once; a set where repetitions are allowed is sometimes called a **bag** or **multi-set** but is beyond the scope of this discussion.

There are at least two predefined sets which have a clear meaning but are hard to define using any other notation:

Definition 8. The set $\emptyset$, called the **null set** or **empty set**, is the set which contains no elements. Note that $\emptyset$ is a set not an element, one cannot write the empty set as $\{\emptyset\}$ since this is the set with one element, that element being the empty set.

Definition 9. The contents of the set $\mathcal{U}$, called the **universal set**, depends on the context. Roughly speaking, it contains every element from the problem being considered.

As a side note, since the elements in a set can be anything we want they can potentially be other sets. Russell's paradox, discovered by mathematician Bertrand Russell in 1901, is a problem with formal set theory that results from this fact. The paradox is similar to the liar paradox seen earlier and is easily stated by considering A, the set of all sets which do not contain themselves. The question is, does A contain itself ? If it does, it should not be in A by definition but it is; if it does not, it should be in the set A by definition but it is not.

1.2.1 Construction

The above method of definition is fine for small finite sets but when the set is large or even infinite in size, writing down all the elements quickly becomes an unpleasant task ! Where there is a natural sequence to the elements, we write continuation dots to save time and space. For example, the same set A as above might be defined as

$$A = \{2, 3, \ldots, 7, 8\}.$$

This set is finite since there is a beginning and an end; the continuation dots are simply a shortcut. As another example, consider the set of numbers exactly divisible by two that might be defined as

$$C = \{2, 4, 6, 8, \ldots\}.$$

Clearly this set is infinite in size in the sense there is no end to the sequence. In this case the continuation dots are a necessity in order to define the set adequately.

The second way to write down sets is using what is sometimes called set builder notation. Basically speaking, we generate the elements in the set using f, a predicate function:

$$D = \{x : f(x)\}.$$

One should read this as "all elements $x \in \mathcal{U}$ such that the predicate $f(x)$ is true". Using set builder notation we can define sets in a more programmatic manner, for example

$$A = \{x : 2 \leq x \leq 8\},$$

and

$$C = \{x : x > 0 \wedge x \equiv 0 \pmod{2}\}$$

define the same sets as we explicitly wrote down above.

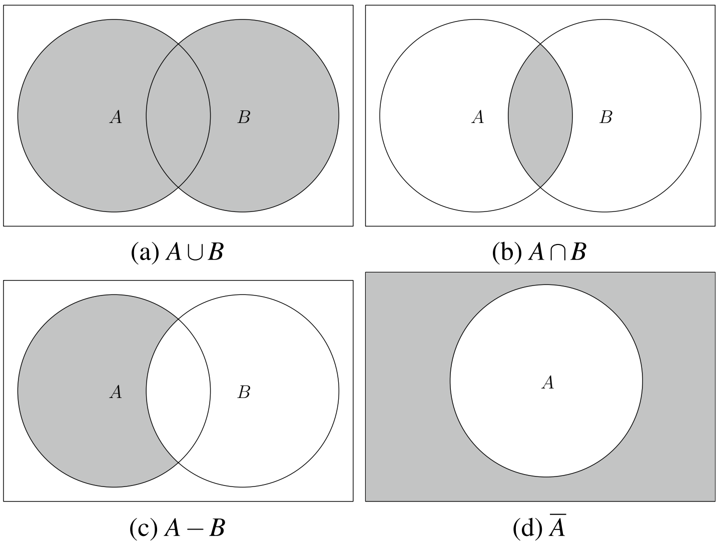


Figure 1.1 A collection of Venn diagrams for standard set operations.

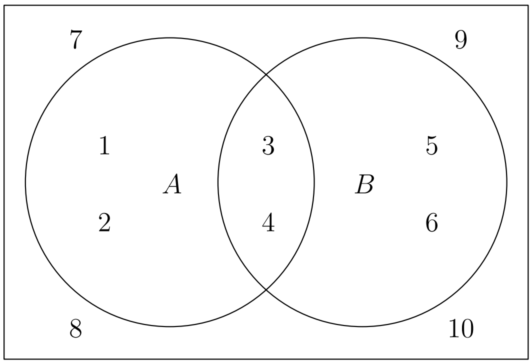


Figure 1.2 An example Venn diagram showing membership of two sets.

1.2.2 Operations

Definition 10. A **sub-set**, say B, of a set A is such that for every $x \in B$ we have that $x \in A$. This is denoted $B \subseteq A$. Conversely, we can say A is a **super-set** of B and write $A \supseteq B$.

Note that every set is a sub-set and super-set of itself and that $A = B$ only if $A \subseteq B$ and $B \subseteq A$. If $A \neq B$, we use the terms **proper sub-set** and **proper super-set** and write $B \subset A$ and $B \supset A$ respectively.

Definition 11. For sets A and B, we have that

- The **union** of A and B is $A \cup B = \{x : x \in A \vee x \in B\}$.
- The **intersection** of A and B is $A \cap B = \{x : x \in A \wedge x \in B\}$.
- The **difference** of A and B is $A - B = \{x : x \in A \wedge x \notin B\}$.
- The **complement** of A is $\overline{A} = \{x : x \in \mathscr{U} \wedge x \notin A\}$.

We say A and B are **disjoint** or **mutually exclusive** if $A \cap B = \emptyset$. Note also that the complement operation can be re-written $A - B = A \cap \overline{B}$.

Definition 12. The power set of a set A, denoted $\mathscr{P}(A)$, is the set of every possible sub-set of A. Note that $\emptyset$ is a member of all power sets.

On first reading, these formal definitions can seem a bit abstract and slightly scary. However, we have another tool at our disposal which describes what they mean in a visual way. This tool is the **Venn diagram**, named after mathematician John Venn who invented the concept in 1881. The basic idea is that sets are represented by regions inside an enclosure that implicitly represents the universal set $\mathscr{U}$. By placing these regions inside each other and overlapping their boundaries, we can describe most set-related concepts very easily.

Figure 1.1 details four Venn diagrams which describe how the union, intersection, difference and complement operations work. The shaded areas of each Venn diagram represent the elements which are in the resulting set. For example, in the diagram for $A \cup B$ the shaded area covers all of the sets A and B: the result contains all elements in either A or B or both. As a simple concrete example, consider the sets

$$\begin{aligned} A &= \{1,2,3,4\} \\ B &= \{3,4,5,6\} \end{aligned}$$

where the universal set is

$$\mathscr{U} = \{1,2,3,4,5,6,7,8,9,10\}.$$

Figure 1.2 shows membership in various settings; recall that those elements within a given region are members of that set. Firstly, we can take the union of A and B as $A \cup B = \{1,2,3,4,5,6\}$ which contains all the elements which are either members of A or B or both. Note that elements 3 and 4 do not appear twice in the result. The intersection of A and B can be calculated as $A \cap B = \{3,4\}$ since these are the elements that are members of both A and B. The difference between A and B, that is the elements in A that are not in B, is $A - B = \{1,2\}$. Finally, the complement of A is all numbers which are not in A, that is $\overline{A} = \{5,6,7,8,9,10\}$.

The union and intersection operations preserve a law of cardinality called the **principle of inclusion** in the sense that we can calculate the cardinality of the output from the cardinality of the inputs as

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

This property is intuitively obvious since those elements in both A and B will be counted twice and hence need subtraction via the last term. We can even check it: in our example above $|A| = 4$ and $|B| = 4$. Checking our results we have that $|A \cup B| = 6$ and $|A \cap B| = 2$ and so by the principle of inclusion we should have $6 = 4 + 4 - 2$ which makes sense.

1.2.3 Numeric Sets

Using this basic notation, we can define three important numeric sets which are used extensively later on in this chapter.

Definition 13. The **integers** are whole numbers which can be positive or negative and also include zero

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}.$$

The **natural numbers** are whole numbers which are positive

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

The **rational numbers** are those which can be expressed in the form p/q where p and q are integers called the **numerator** and **denominator**

$$\mathbb{Q} = \{p/q : p \in \mathbb{Z} \wedge q \in \mathbb{Z} \wedge q \neq 0\}.$$

Clearly the set of rational numbers is a super-set of both $\mathbb{Z}$ and $\mathbb{N}$ since, for example, we can write $p/1$ to represent any integer p as a member of $\mathbb{Q}$. However, not all numbers are rational. Some are **irrational** in the sense that it is impossible to find a p and q such that they exactly represent the required result; examples include the value of π.

1.2.4 Functions

Definition 14. If A and B are sets, a **function** f from A to B is a process that maps each element of A to an element of B. We write this as

$$f : A \to B$$

where A is termed the **domain** of f and B is the **codomain** of f. For an element $x \in A$, which we term the **preimage**, there is only one $y = f(x) \in B$ which is termed the **image** of x. Finally, the set

$$\{y : y = f(x) \wedge x \in A \wedge y \in B\}$$

which is all possible results, is termed the **range** of f and is always a sub-set of the **codomain**.

As a concrete example, consider a function $\textsc{Inv}$ which takes an integer x as input and produces the rational number $1/x$ as output:

$$\textsc{Inv} : \mathbb{Z} \to \mathbb{Q}, \;\; \textsc{Inv}(x) = 1/x.$$

Note that here we write the **function signature** which defines the domain and codomain of $\textsc{Inv}$ inline with the definition of the **function behaviour**. In this case the domain of $\textsc{Inv}$ is $\mathbb{Z}$ since it takes integers as input; the codomain is $\mathbb{Q}$ since it produces rational numbers as output. If we take an integer and apply the function to get something like $\textsc{Inv}(2) = 1/2$, we have that $1/2$ is the image of 2 or conversely 2 is the preimage of $1/2$ under $\textsc{Inv}$.

From this definition it might seem as though we can only have functions with one input and one output. However, remember that we are perfectly entitled to have sets of sets so we can easily define a function f, for example, as

$$f : A \times A \to B.$$

This function takes elements from the Cartesian product of A as input and produces an element of B as output. So since pairs $(x,y) \in A \times A$ are used as input, f can in some sense accept two input values. As a concrete example consider the function

$$\textsc{Max} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \;\; \textsc{Max}(x,y) = \begin{cases} x & \text{if} \;\; x > y \\ y & \text{otherwise.} \end{cases}$$

This is the maximum function on integers; it takes two integers as input and produces an integer, the maximum of the inputs, as output. So if we take a pair of integers, say $(2,4)$, and apply the function we get $\textsc{Max}(2,4) = 4$ where we usually omit the parentheses around the pair of inputs. In this case, the domain of $\textsc{Max}$ is $\mathbb{Z} \times \mathbb{Z}$ and the codomain is $\mathbb{Z}$; the integer 4 is the image of the pair $(2,4)$ under $\textsc{Max}$.

Definition 15. For a given function f, we say that f is

- **surjective** if the range equals the codomain, i.e., there are no elements in the codomain which do not have a preimage in the domain.
- **injective** if no two elements in the domain have the same image in the range.
- **bijective** if the function is both surjective and injective, i.e., every element in the domain is mapped to exactly one element in the codomain.

Using the examples above, we clearly have that $\textsc{Inv}$ is not surjective but $\textsc{Max}$ is. This follows because we can construct a rational $2/3$ which does not have an integer preimage under $\textsc{Inv}$ so the function cannot be surjective. Equally, for any integer x in the range of $\textsc{Max}$ there is always a pair (x,y) in the domain such that $x > y$ so

$\textsc{Max}$ is surjective, in fact there are lots of them since $\mathbb{Z}$ is infinite in size ! In the same way, we have that $\textsc{Inv}$ is injective but $\textsc{Max}$ is not. Only one preimage x maps to the value $1/x$ in the range under $\textsc{Inv}$ but there are multiple pairs (x,y) which map to the same image under $\textsc{Max}$, for example 4 is the image of both $(1,4)$ and $(2,4)$ under $\textsc{Max}$.

Definition 16. Given two functions $f : A \to B$ and $g : B \to C$, the **composition** of f and g is denoted

$$g \circ f : A \to C.$$

Given some input $x \in A$, this composition is equivalent to applying $y = f(x)$ and then $z = g(y)$ to get the result $z \in C$. More formally, we have

$$(g \circ f)(x) = g(f(x)).$$

The notation $g \circ f$ should be read as "apply g to the result of applying f".

Definition 17. The **identity function** $\mathscr{I}$ on a set A is defined by

$$\mathscr{I} : A \to A, \;\; \mathscr{I}(x) = x$$

so that it maps all elements to themselves. Given two functions f and g defined by $f : A \to B$ and $g : B \to A$, if $g \circ f$ is the identity function on set A and $f \circ g$ is the identity on set B, then f is the **inverse** of g and g is the inverse of f. We denote this by $f = g^{-1}$ and $g = f^{-1}$. If a function f has an inverse, we hence have $f^{-1} \circ f = \mathscr{I}$.

The inverse of a function maps elements from the codomain back into the domain, essentially reversing the original function. It is easy to see not all functions have an inverse. For example, if the function is not injective then there will be more than one potential preimage for the inverse of any image.

At first glance, it might seem like our example functions both have inverses but they do not. For example, given some value $1/x$, we can certainly find x but we have already said that numbers like $2/3$ also exist in the codomain so we cannot invert all the values we might come across. However, consider the example of a successor function on integers

$$\textsc{Succ} : \mathbb{Z} \to \mathbb{Z}, \;\; \textsc{Succ}(x) = x + 1$$

which takes an integer x as input and produces $x + 1$ as output. The function is bijective since the codomain and range are the same and no two integers have the same successor. Thus we have an inverse and it is easy to describe as

$$\textsc{Pred} : \mathbb{Z} \to \mathbb{Z}, \;\; \textsc{Pred}(x) = x - 1$$

which is the predecessor function: it takes an integer x as input and produces $x - 1$ as output. To see that $\textsc{Succ}^{-1} = \textsc{Pred}$ and $\textsc{Succ}^{-1} = \textsc{Pred}$ note that

$$(\textsc{Pred} \circ \textsc{Succ})(x) = (x + 1) - 1 = x$$

which is the identity function. Conversely,

$$(\textsc{Succ} \circ \textsc{Pred})(x) = (x-1)+1 = x$$

which is also the identity function.

1.2.5 Relations

Definition 18. We call a sequence of n elements $(x_0, x_1, \ldots, x_{n-1})$ an **n-tuple** or simply a **tuple** when the number of elements is irrelevant. In the specific case of $n = 2$, we call (x_0, x_1) a **pair**. The i-th element of the tuple x is denoted x_i, and the number of elements in a particular tuple x may be written as $|x|$.

Definition 19. The **Cartesian product** of n sets, say $A_0, A_1, \ldots, A_{n-1}$, is defined as

$$A_0 \times A_1 \times \cdots \times A_{n-1} = \{(a_0, a_1, \ldots, a_{n-1}) : a_0 \in A_0, a_1 \in A_1, \ldots, a_{n-1} \in A_{n-1}\}.$$

In the most simple case of $n = 2$, the Cartesian product $A_0 \times A_1$ is the set of all possible pairs where the first item in the pair is a member of A_0 and the second item is a member of A_1.

The Cartesian product of a set A with itself n times is denoted A^n. To be complete, we define $A^0 = \emptyset$ and $A^1 = A$. Finally, by writing A^* we mean the Cartesian product of A with itself a finite number of times.

Cartesian products are useful to us since they allow easy description of sequences, or **vectors**, of elements. Firstly, we often use them to describe vectors of elements from a set. So for example, say we have the set of digits $A = \{0, 1\}$. The Cartesian product of A with itself is

$$A \times A = \{(0,0), (0,1), (1,0), (1,1)\}.$$

If you think of the pairs as more generally being vectors of these digits, the Cartesian product A^n is the set of all possible vectors of 0 and 1 which are n elements long.

Definition 20. Informally, a **binary relation** f on a set A is like a predicate function which takes members of the set as input and "filters" them to produce an output. As a result, for a set A the relation f forms a sub-set of $A \times A$. For a given set A and a binary relation f, we say f is

- **reflexive** if $f(x,x) = \text{true}$ for all $x \in A$.
- **symmetric** if $f(x,y) = \text{true}$ implies $f(y,x) = \text{true}$ for all $x, y \in A$.
- **transitive** if $f(x,y) = \text{true}$ and $f(y,z) = \text{true}$ implies $f(x,z) = \text{true}$ for all $x, y, z \in A$.

If f is reflexive, symmetric and transitive, then we call it an **equivalence relation**.

The easiest way to think about this is to consider a concrete example such as the case where our set is $A = \{1,2,3,4\}$ such that the Cartesian product is

$$A \times A = \left\{ \begin{array}{llll} (1,1), & (1,2), & (1,3), & (1,4), \\ (2,1), & (2,2), & (2,3), & (2,4), \\ (3,1), & (3,2), & (3,3), & (3,4), \\ (4,1), & (4,2), & (4,3), & (4,4) \end{array} \right\}.$$

Imagine we define a function

$$\text{EQU} : \mathbb{Z} \times \mathbb{Z} \to \{\text{true}, \text{false}\}, \text{EQU}(x,y) = \begin{cases} \text{true} & \text{if } x = y \\ \text{false} & \text{otherwise} \end{cases}$$

which tests whether two inputs are equal. Using the function we can form a sub-set of $A \times A$ (called A_{EQU} for example) in the sense that we can pick out the pairs (x,y)

$$A_{\text{EQU}} = \{(1,1),(2,2),(3,3),(4,4)\}$$

where $\text{EQU}(x,y) = \text{true}$. For members of A, say $x,y,z \in A$, we have that $\text{EQU}(x,x) = \text{true}$ so the relation is reflexive. If $\text{EQU}(x,y) = \text{true}$, then $\text{EQU}(y,x) = \text{true}$ so the relation is also symmetric. Finally, if $\text{EQU}(x,y) = \text{true}$ and $\text{EQU}(y,z) = \text{true}$, then we must have that $\text{EQU}(x,z) = \text{true}$ so the relation is also transitive and hence an equivalence relation.

Now imagine we define another function

$$\text{LTH} : \mathbb{Z} \times \mathbb{Z} \to \{\text{true}, \text{false}\}, \text{LTH}(x,y) = \begin{cases} \text{true} & \text{if } x < y \\ \text{false} & \text{otherwise} \end{cases}$$

which tests whether one input is less than another, and consider the sub-set of A

$$A_{\text{LTH}} = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$

of all pairs (x,y) with $x,y \in A$ such that $\text{LTH}(x,y) = \text{true}$. It cannot be that $\text{LTH}(x,x) = \text{true}$, so the relation is not reflexive; we say it is irreflexive. It also cannot be that if $\text{LTH}(x,y) = \text{true}$ then $\text{LTH}(y,x) = \text{true}$, so the relation is also not symmetric; it is anti-symmetric. However, if both $\text{LTH}(x,y) = \text{true}$ and $\text{LTH}(y,z) = \text{true}$, then $\text{LTH}(x,z) = \text{true}$ so the relation is transitive.

1.3 Boolean Algebra

Most people are taught about simple numeric algebra in school. One has

- a set of values, say $\mathbb{Z}$,
- a list of operations, say $-$, $+$ and $\cdot$,
- and a list of axioms which dictate how $-$, $+$ and $\cdot$ work.

You might not know the names for these axioms but you probably know *how* they work, for example you probably know for some $x, y, z \in \mathbb{Z}$ that $x + (y + z) = (x + y) + z$.

In the early 1840s, mathematician George Boole combined propositional logic and set theory to form a system which we now call Boolean algebra [4]. Put (rather too) simply, Boole saw that working with a logic expression x is much the same as working with a number y. One has

- a set of values, $\{\text{false}, \text{true}\}$,
- a list of operations $\neg$, $\vee$ and $\wedge$,
- and a list of axioms that dictate how $\neg$, $\vee$ and $\wedge$ work.

Ironically, at the time his work was viewed as somewhat obscure and indeed Boole himself did not necessarily regard logic directly as a mathematical concept. However, the step of finding common themes within two such concepts and unifying them into one has been a powerful tool in mathematics, allowing more general thinking about seemingly diverse objects. It was not until 1937 that Claude Shannon, then a student of both electrical engineering and mathematics, saw the potential of using Boolean algebra to represent and manipulate digital information [60].

Definition 21. A **binary operation** $\odot$ on some set A is a function

$$\odot : A \times A \rightarrow A$$

while a **unary operation** $\oslash$ on A is a function

$$\oslash : A \rightarrow A.$$

Definition 22. Define a set $\mathbb{B} = \{0, 1\}$ on which there are two binary operators $\wedge$ and $\vee$ and a unary operator $\neg$. The members of $\mathbb{B}$ act as **identity elements** for $\wedge$ and $\vee$. These operators and identities are governed by a number of axioms which should look familiar:

equivalence	$x \leftrightarrow y$	$\equiv (x \rightarrow y) \wedge (y \rightarrow x)$
implication	$x \rightarrow y$	$\equiv \neg x \vee y$
involution	$\neg\neg x$	$\equiv x$
idempotency	$x \wedge x$	$\equiv x$
commutativity	$x \wedge y$	$\equiv y \wedge x$
association	$(x \wedge y) \wedge z$	$\equiv x \wedge (y \wedge z)$
distribution	$x \wedge (y \vee z)$	$\equiv (x \wedge y) \vee (x \wedge z)$
de Morgan's	$\neg(x \wedge y)$	$\equiv \neg x \vee \neg y$
identity	$x \wedge 1$	$\equiv x$
null	$x \wedge 0$	$\equiv 0$
inverse	$x \wedge \neg x$	$\equiv 0$
absorption	$x \wedge (x \vee y)$	$\equiv x$
idempotency	$x \vee x$	$\equiv x$
commutativity	$x \vee y$	$\equiv y \vee x$
association	$(x \vee y) \vee z$	$\equiv x \vee (y \vee z)$
distribution	$x \vee (y \wedge z)$	$\equiv (x \vee y) \wedge (x \vee z)$
de Morgan's	$\neg(x \vee y)$	$\equiv \neg x \wedge \neg y$
identity	$x \vee 0$	$\equiv x$
null	$x \vee 1$	$\equiv 1$
inverse	$x \vee \neg x$	$\equiv 1$
absorption	$x \vee (x \wedge y)$	$\equiv x$

We call the $\wedge$, $\vee$ and $\neg$ operators NOT, AND and OR respectively.

Notice how this description of a Boolean algebra unifies the concepts of propositional logic and set theory: 0 and false and $\emptyset$ are sort of equivalent, as are 1 and true and $\mathscr{U}$. Likewise, $x \wedge y$ and $A \cap B$ are sort of equivalent, as are $x \vee y$ and $A \cup B$ and $\neg x$ and $\overline{A}$: you can even draw Venn diagrams to illustrate this fact. It might seem weird to see the statement

$$x \vee x = x$$

written down but we know what it means because the axioms allow one to manipulate the statement as follows:

$$\begin{aligned} x \vee x &= (x \vee x) \wedge 1 && \text{(identity)} \\ &= (x \vee x) \wedge (x \vee \neg x) && \text{(inverse)} \\ &= x \vee (x \wedge \neg x) && \text{(distribution)} \\ &= x \vee 0 && \text{(inverse)} \\ &= x && \text{(identity)} \end{aligned}$$

Also notice that the $\wedge$ and $\vee$ operations in a Boolean algebra behave in a similar way to $\cdot$ and $+$ in a numerical algebra; for example we have that $x \vee 0 = x$ and $x \wedge 0 = 0$. As such, $\wedge$ and $\vee$ are often termed (and written as) the "product" and "sum" operations.

1.3.1 Boolean Functions

Definition 23. In a given Boolean algebra over the set $\mathbb{B}$, a Boolean function f can be described as

$$f : \mathbb{B}^n \to \mathbb{B}.$$

That is, it takes an n-tuple of elements from the set as input and produces an element of the set as output.

This definition is more or less the same as the one we encountered earlier. For example, consider a function f that takes two inputs whose signature we can write as

$$f : \mathbb{B}^2 \to \mathbb{B}$$

so that for $x, y, z \in \mathbb{B}$, the input is a pair (x, y) and the output for a given x and y is written $z = f(x, y)$. We can define the actual mapping performed by f in two ways. Firstly, and similarly to the truth tables seen previously, we can simply enumerate the mapping between inputs and outputs:

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

However, with a large number of inputs this quickly becomes difficult so we can also write f as a Boolean expression

$$f : \mathbb{B}^2 \to \mathbb{B}, \;\; f(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y)$$

which describes the operation in a more compact way.

If we require more than one output, say m outputs, from a given function, we can think of this just m separate functions each of which produces one output from the same inputs. For example, taking $m = 2$ a function with the signature

$$g : \mathbb{B}^2 \to \mathbb{B}^2$$

can be split into two functions

$$\begin{array}{l} g_0 : \mathbb{B}^2 \to \mathbb{B} \\ g_1 : \mathbb{B}^2 \to \mathbb{B} \end{array}$$

so that the outputs of g_0 and g_1 are grouped together to form the output of g. We can again specify the functions either with a truth table

x	y	$g_0(x,y)$	$g_1(x,y)$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

or a pair of expressions

$$\begin{array}{l} g_0 : \mathbb{B}^2 \rightarrow \mathbb{B}, \;\; g_0(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y) \\ g_1 : \mathbb{B}^2 \rightarrow \mathbb{B}, \;\; g_1(x,y) = (x \wedge y) \end{array}$$

such that the original function g can be roughly described as

$$g(x,y) = (g_0(x,y), g_1(x,y)).$$

This end result is often termed a **vectorial** Boolean function: the inputs and outputs are vectors over the set $\mathbb{B}$ rather than single elements of the set.

1.3.2 Normal Forms

Definition 24. Consider a Boolean function f with n inputs, for example $f(x_0, x_1, \ldots, x_{n-1})$.

When the expression for f is written as a sum (i.e., OR) of terms which each comprise the product (i.e., AND) of a number of inputs, it is said to be in **disjunctive normal form** or **Sum of Products (SoP)** form; the terms in this expression are called the **minterms**. Conversely, when the expression for f is written as a product (i.e., AND) of terms which each comprise the sum (i.e., OR) of a number of inputs, it is said to be in **conjunctive normal form** or **Product of Sums (PoS)** form; the terms in this expression are called the **maxterms**.

This is a formal definition for something very simple to describe by example. Consider a function f which is described by

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

The minterms are the second and third rows of this table, the maxterms are the first and fourth lines. An expression for f in SoP form is

$$f_{SoP}(x,y) = (\neg x \wedge y) \vee (x \wedge \neg y).$$

Notice how the terms $\neg x \wedge y$ and $x \wedge \neg y$ are the minterms of f: when the term is 1 or 0, the corresponding minterm is 1 or 0. It is usually crucial that all the variables appear in all the minterms so that the function is exactly described. To see why this is so, consider writing an incorrect SoP expression by removing the reference to y from the first minterm so as to get

$$(\neg x) \vee (x \wedge \neg y).$$

Since $\neg x$ is 1 for the first and second rows, rather than just the second as was the case with $\neg x \wedge y$, we have described another function $f' \neq f$ with the truth table

x y	$f'(x,y)$
0 0	1
0 1	1
1 0	1
1 1	0

In a similar way to above, we can construct a PoS expression for f as

$$f_{PoS}(x,y) = (x \vee y) \wedge (\neg x \vee \neg y).$$

In this case the terms $x \vee y$ and $\neg x \vee \neg y$ are the maxterms. To verify that the two forms represent the same function, we can apply de Morgan's axiom to get

$$\begin{aligned} f_{PoS}(x,y) &= (x \vee y) \wedge (\neg x \vee \neg y) \\ &= (\neg x \wedge \neg y) \vee (x \wedge y) \text{ (de Morgan's).} \end{aligned}$$

Although this might not seem particularly interesting at first glance, it gives us quite a powerful technique. Given a truth table for an arbitrary function, we can construct an expression for the function simply by extracting either the minterms or maxterms and constructing standard SoP or PoS forms.

1.4 Number Systems

Definition 25. One can write an integer x as

$$x = \pm \sum_{i=0}^{n-1} x_i \cdot b^i$$

where each x_i is taken from the **digit set** $\mathbb{D} = \{0 \ldots b-1\}$. This is called the base-b **expansion** of x; the selection of b determines the **base** or **radix** of the number system x is written in. The same integer can be written as

$$x = \pm(x_0, x_1, \ldots, x_{n-1})^{(b)}$$

which is called the base-b vector representation of x.

Definition 26. We use $x_{(b)}$ to denote the integer literal x in base-b. Where it is not clear, we use $|x_{(b)}|$ to denote n, the number of digits in the base-b expansion of x. At times this might look a little verbose or awkward, however it allows us to be exact at all times about the meaning of what we are writing.

We are used to working in the **decimal** or **denary** number system by setting $b = 10$, probably because we (mostly) have ten fingers and toes. In this case, we have that $x_i \in \{0 \ldots 9\}$ with each digit representing a successive power of ten. Using this number system we can easily write down integers such as

$$123_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Note that for all y, we define $y^0 = 1$ and $y^1 = y$ so that in the above example $10^0 = 1$ and $10^1 = 10$. Furthermore, note that we can also have negative values of i so, for example, $10^{-1} = 1/10$ and $10^{-2} = 1/100$. Using this fact, we can also write down rational numbers such as

$$123.3125_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 3 \cdot 10^{-1} + 1 \cdot 10^{-2} + 2 \cdot 10^{-3} + 5 \cdot 10^{-4}.$$

Of course, one can select other values for b to represent x in different bases. Common choices are $b = 2$, $b = 8$ and $b = 16$ which form the **binary**, **octal** and **hexadecimal** number systems. Irrespective of the base, integer and rational numbers are written using exactly the same method so that

$$\begin{aligned} 123_{(10)} &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1111011_{(2)} \end{aligned}$$

and

$$\begin{aligned} 123.3125_{(10)} &= 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-2} + 1 \cdot 2^{-4} \\ &= 1111011.0101_{(2)}. \end{aligned}$$

The **decimal point** we are used to seeing is termed the **binary point** in this case. Considering rational numbers, depending on the base used there exist some values which are not exactly representable. Informally, this is obvious since when we write $1/3$ as a decimal number we have $0.3333r_{(10)}$ with the appended letter r meaning that the sequence recurs infinitely. Another example is $1/10$ which can be written as $0.1_{(10)}$ but is not exactly representable in binary, the closest approximation being $0.000110011r_{(2)}$.

There are several properties of the binary expansion of numbers that warrant a brief introduction. Firstly, the number of digits set to 1 in a binary expansion of some value x is called the **Hamming weight** of x after Richard Hamming, a researcher at Bell Labs. Hamming also introduced the idea of the distance between two values which captures how many digits differ; this is named the **Hamming distance**. For example, consider the two values

$$\begin{aligned} x &= \ \ 9_{(10)} = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1001_{(2)} \\ y &= 14_{(10)} = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 1110_{(2)}. \end{aligned}$$

The Hamming weights of x and y, say $H(x)$ and $H(y)$, are 2 and 3 respectively; the Hamming distance between x and y is 3 since three bits differ.

Using the hexadecimal numbers system is slightly complicated by the fact we are used to only using the ten decimal digits $0 \ldots 9$ yet need to cope with sixteen hexadecimal digits. As a result, we use the letters $A \ldots F$ to write down the base-10 numbers $10 \ldots 15$. This means we have

$$\begin{aligned} 7B_{(16)} &= 7 \cdot 16^1 + \ B \cdot 16^0 \\ &= 7 \cdot 16^1 + 11 \cdot 16^0 \\ &= 123_{(10)} \qquad\qquad . \end{aligned}$$

Each octal or hexadecimal digit represents exactly three or four binary digits respectively; using this shorthand one can more easily write and remember long vectors of binary digits.

Aside from the issue of not being able to exactly represent some rational numbers in some bases, it is crucial to realise that an actual number x and how we represent it are largely independent. For example we can write any of

$$\begin{aligned} 123_{(10)} &= 1111011_{(2)} \\ &= 173_{(8)} \\ &= 7B_{(16)} \end{aligned}$$

yet they all represent the same number, they still mean the same thing, exactly the same rules apply in all bases even if they seem unnatural or unfamiliar. The choice of base then is largely one of convenience in a given situation, so we really want to select a base suitable for use in the context of digital computers and computation.

1.4.1 Converting Between Bases

Considering just integers for the moment, converting between different bases is a fairly simple task. Algorithm 1.1 demonstrates how it can be achieved; clearly we need to be able to do arithmetic in both bases for it to be practical. Simply put, we extract one digit of the result at a time. As an example run, consider converting $123_{(10)}$ into binary, i.e., $x = 123$, $\hat{b} = 10$ and $\bar{b} = 2$:

Input: An integer number x represented in base-$\hat{b}$.
Output: The same number represented in base-$\bar{b}$.

```
INTEGER-BASE-CONVERT($x, \hat{b}, \bar{b}$)
    $i \leftarrow 0$
    $t \leftarrow x$
    if $x < 0$ then
        $t \leftarrow -t$
    $r \leftarrow 0$
    while $t \neq 0$ do
        $m \leftarrow t \bmod \bar{b}$
        $t \leftarrow \lfloor t/\bar{b} \rfloor$
        $r \leftarrow r + (m \cdot \bar{b}^i)$
        $i \leftarrow i + 1$
    if $x < 0$ then
        $r \leftarrow -r$
    return $r$
end
```

Algorithm 1.1: An algorithm for converting integers from representation in one base to another.

Input: A rational number x represented in base-$\hat{b}$, an integer d determining the number of digits required.
Output: The same number represented in base-$\bar{b}$.

```
FRACTION-BASE-CONVERT($x, \hat{b}, \bar{b}, d$)
    $i \leftarrow 1$
    $t \leftarrow x$
    if $x < 0$ then
        $t \leftarrow -t$
    $r \leftarrow 0$
    while $t \neq 0$ and $i < d$ do
        $m \leftarrow t \cdot \bar{b}$
        $n \leftarrow \lfloor m \rfloor$
        $r \leftarrow r + (n \cdot \bar{b}^{-(i+1)})$
        $t \leftarrow m - n$
        $i \leftarrow i + 1$
    return $r$
end
```

Algorithm 1.2: An algorithm for converting the fractional part of a rational number from one base to another.

i	m	t	r
$-$	$-$	$123_{(10)}$	$0000000_{(2)} = 0$
0	1	$61_{(10)}$	$0000001_{(2)} = 2^0$
1	1	$30_{(10)}$	$0000011_{(2)} = 2^0 + 2^1$
2	0	$15_{(10)}$	$0000011_{(2)} = 2^0 + 2^1$
3	1	$7_{(10)}$	$0001011_{(2)} = 2^0 + 2^1 + 2^3$
4	1	$3_{(10)}$	$0011011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4$
5	1	$1_{(10)}$	$1111011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4 + 2^5$
6	1	$0_{(10)}$	$1111011_{(2)} = 2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6$

One can see how the result is built up a digit at a time by finding the remainder of x after division by the target base. Finally, if the input was negative we negate the accumulated result and return it.

Dealing with rational numbers is more tricky. Clearly we can still convert the integer part of such a number using Algorithm 1.1. The problem with using a similar approach with the fractional part is that if the number cannot be represented exactly in a given base, our loop to generate digits will never terminate. To cope with this Algorithm 1.2 takes an extra parameter which determines the number of digits we require. As another example, consider the conversion of $0.3125_{(10)}$ into four binary digits, i.e., $x = 0.3125$, $\hat{b} = 10$, $\bar{b} = 2$ and $d = 4$:

i	n	t	r
$-$	$-$	$0.3125_{(10)}$	$0.0000_{(2)} = 0$
1	1	$0.6250_{(10)}$	$0.0100_{(2)} = 2^{-2}$
2	0	$0.2500_{(10)}$	$0.0100_{(2)} = 2^{-2}$
3	1	$0.5000_{(10)}$	$0.0101_{(2)} = 2^{-2} + 2^{-4}$

On each step, we move one digit across to the left of the fractional point by multiplying by the target base. This digit is extracted using the floor function; $\lfloor x \rfloor$ denotes the integer part of a rational number x. Then, so we can continue processing the rest of the number, we remove the digit to the left of the fractional point by subtracting it from our working accumulator. We already set the sign of our number when converting the integer part of the input so we do not need to worry about that.

1.4.2 Bits, Bytes and Words

Concentrating on binary numbers, a single binary digit is usually termed a **bit**. To represent numbers we group together bits into larger sequences; we usually term these **binary vectors**.

Definition 27. An n-bit **binary vector** or **bit-vector** is a member of the set $\mathbb{B}^n$ where $\mathbb{B} = \{0, 1\}$, i.e., it is an n-tuple of bits. We use x_i to denote the i-th bit of the binary vector x and $|x| = n$ to denote the number of bits in x. Instead of writing out $x \in \mathbb{B}^n$ fully as the n-tuple $(x_0, x_1, \ldots, x_{n-1})$ we typically just list the digits in sequence.

For example, consider the following bit-vector

$$(1,1,0,1,1,1,1)$$

which can be less formally written as

$$1111011$$

and has seven bits in it. Note that there is no notation for what base the number is in and so on: this is not a mistake. This is just a vector of binary digits, in one context it could be representing a number but in another context perhaps it represents part of an image: there is a clear distinction between a number x and the representation of x as a bit-vector.

Definition 28. We use the **concatenation** operator (written as $x : y$) to join together two or more bit-vectors; for example $111 : 1011$ describes the bit-vector 1111011.

We do however need some standard way to make sure we know which bits in the vector represent which bits from the more formal n-tuple; this is called an **endianness** convention. Usually, and as above, we use a **little-endian** convention by reading the bits from right to left, giving each bit is given an index. So writing the number $1111011_{(2)}$ as the vector 1111011 we have that

$$\begin{aligned}
x_0 &= 1 \\
x_1 &= 1 \\
x_2 &= 0 \\
x_3 &= 1 \\
x_4 &= 1 \\
x_5 &= 1 \\
x_6 &= 1
\end{aligned}$$

If we interpret this vector as a binary number, it therefore represents $1111011_{(2)} = 123_{(10)}$. In this case, the right-most bit (bit zero or x_0) is termed the **least-significant** while the left-most bit (bit $n-1$ or x_{n-1}) is the **most-significant**. However, there is no reason why little-endian this is the only option: a **big-endian** naming convention reverses the indices so we now read them left to right. The left-most bit (bit zero or x_0) is now the most-significant and the right-most (bit $n-1$ or x_{n-1}) is the least-significant. If the same vector is interpreted in big-endian notation, we find that

$$\begin{aligned} x_0 &= 1 \\ x_1 &= 1 \\ x_2 &= 1 \\ x_3 &= 1 \\ x_4 &= 0 \\ x_5 &= 1 \\ x_6 &= 1 \end{aligned}$$

and if interpreted as a binary number, the value now represents $1101111_{(2)} = 111_{(10)}$.

We have special terms for certain length vectors. A vector of eight bits is termed a **byte** (or octet) while a vector of four bits is a **nibble** (or nybble). A given context will, in some sense, have a natural value for n. Vectors of bits which are of this natural size are commonly termed **words**; one may also see **half-word** and **double-word** used to describe $(n/2)$-bit and $2n$-bit vectors. The word size is a somewhat vague concept but becomes clear with some context. For example, when you hear a computer processor described as being 32-bit or 64-bit this is, roughly speaking, specifying that the word size is 32 or 64 bits. Nowadays, the word size of a processor is normally a power-of-two since this simplifies a number of issues. But this was not always the case: the EDSAC computer used a 17-bit word since this represented a limit in terms of expenditure on hardware !

The same issues of endianness need consideration in the context of bytes and words. For example, one can regard a 32-bit word as consisting of four 8-bit bytes. Taking W, X, Y and Z as bytes, we can see that the word can be constructed two ways. Using a little-endian approach, Z is byte zero, Y is byte one, X is byte two and W is byte three; we have the word constructed as $W : X : Y : Z$. Using a big-endian approach the order is swapped so that Z is byte three, Y is byte two, X is byte one and W is byte zero; we have the word constructed as $Z : Y : X : W$.

1.4.3 Representing Numbers and Characters

The sets $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{Q}$ are infinite in size. But so far we have simply written down such numbers; our challenge in using them on a computer is to represent the numbers using finite resources. Most obviously, these finite resources place a bound on the number of digits we can have in our base-b expansion.

We have already mentioned that modern digital computers prefer to work in binary. More specifically, we need to represent a given number within n bits: this includes the sign and the magnitude which combine to form the value. In discussing how this is achieved we use the specific case of $n = 8$ by representing given numbers within a byte.

Decimal	BCD Encoding
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 1.1 A table showing a standard BCD encoding system.

1.4.3.1 Representing Integer Numbers

The **Binary Coded Decimal (BCD)** system is one way of encoding decimal digits as binary vectors. Believe it or not, there are several standards for BCD. The most simple of these is called Simple Binary Coded Decimal (SBCD) or BCD 8421: one represents a single decimal digit as a vector of four binary digits according to Table 1.1. So for example, the number $123_{(10)}$ would be represented (with spacing for clarity) by the vector

$$0001\ \ 0010\ \ 0011.$$

Clearly there are some problems with BCD, most notably that there are six unused encodings, $1010 \ldots 1111$, and hence the resulting representation is not very compact. Also, we need something extra at the start of the encoding to tell us if the number is positive or negative. Even so, there are applications for this sort of scheme where decimal arithmetic is de rigueur.

Consigning BCD to the dustbin of history, we can take a much easier approach to representing unsigned integers. We simply let an n-bit binary vector represent our integer as we have already been doing so far. In such a representation an integer x can take the values

$$0 \le x \le 2^n - 1.$$

However, storing signed integers is a bit more tricky since we need some space to represent the sign of the number. There are two common ways of achieving this: the sign-magnitude and twos-complement methods.

The **sign-magnitude** method represents a signed integer in n bits by allocating one bit to store the sign, typically the most-significant, and $n-1$ to store the magnitude. The sign bit is set to one for a negative integer and zero for a positive integer so that x is expressed as

$$x = -1^{x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i.$$

Hence we find that our example is represented in eight bits by

$$01111011 = +1 \cdot (2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0) = +123_{(10)}$$

while the negation is represented by

$$11111011 = -1 \cdot (2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0) = -123_{(10)}.$$

In such an n-bit sign-magnitude representation, an integer x can take the values

$$-2^{n-1} - 1 \leq x \leq +2^{n-1} - 1.$$

For our 8-bit example, this means we can represent $-127 \ldots +127$.

Note that in a sign-magnitude representation there are two versions of zero, namely ± 0. This can make constructing arithmetic a little more tricky than it might otherwise be. As alternative to this approach, the **twos-complement** representation removes the need for a dedicated sign bit and thus the problem of there being two versions of zero. Essentially, twos-complement defines the most-significant bit of an n-bit value to represent -2^{n-1} rather than $+2^{n-1}$. Thus, an integer x is written as

$$x = x_{n-1} \cdot -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i.$$

Starting with the value

$$+123_{(10)} = 01111011$$

we obtain the twos-complement negation as

$$-123_{(10)} = 10000101$$

since

$$-123_{(10)} = 1 \cdot -2^7 + 1 \cdot 2^2 + 1 \cdot 2^0.$$

The range of values in an n-bit twos-complement representation is slightly different than using sign-magnitude; an integer x can represent the values in the range

$$-2^{n-1} \leq x \leq +2^{n-1} - 1.$$

So for our 8-bit example, this means values in the range $-128 \ldots +127$ with the difference from the sign-magnitude version coming as a result of there only being one version of zero: the twos-complement of zero is zero. As you might expect, calculating the twos-complement negation of a number can be done with a simple algorithm; we see how this works when we introduce methods for performing arithmetic with numbers in Chapter 7.

It is crucial to realise that the representation of a number in either method is dependent on the value of n we need to fit them in. For example, setting $n = 4$ the sign-magnitude representation of $-2_{(10)}$ is 1010 while with $n = 8$ it is 10000010. A similar argument is true of twos-complement: it is not a case of just padding zeros onto one end to make it the right size.

1.4.3.2 Representing Characters

Imagine that we want to represent a sequence or **string** of letters or **characters**; perhaps as used in a call to the `printf` function in a C program. We have already seen how to represent integers using binary vectors that are suitable for digital computers, the question is how can we do the same with characters ? Of course, this is easy, we just need to translate from one to the other. More specifically, we need two functions: $\textsc{Ord}(x)$ which takes a character x and gives us back the corresponding integer representation, and $\textsc{Chr}(y)$ which takes an integer representation y and gives us back the corresponding character. But how do we decide how the functions should work ?

Fortunately, people have thought about this and provided standards we can use. One of the oldest, from circa 1967, and most simple is the **American Standard Code for Information Interchange (ASCII)**, pronounced "ass key". ASCII has a rich history but was originally developed to communicate between early "teleprinter" devices. These were like the combination of a typewriter and a telephone: the devices were able to communicate text to each other, before innovations such as the fax machine. Later, but long before monitors and graphics cards existed, similar devices allowed users to send input to early computers and receive output from them. Table 1.2 shows the 128-entry ASCII table which tells us how characters are represented as numbers; 95 are printable characters that we can recognise (including *SPC* which is short for "space"), 33 are non-printable control characters. Originally these would have been used to control the teleprinter rather than to have it print something. For example, the *CR* and *LF* characters (short for "carriage return" and "line feed") would combine to move the print head onto the next line; we still use these characters to mark the end of lines in text files. Other control characters still play a role in more modern computers as well: the *BEL* (short for "bell") characters play a "ding" sound when printed to most UNIX terminals, we have keyboards with keys that relate to *DEL* and *ESC* (short for "delete" and "escape") and so on.

Given the ASCII table, we can see that for example $\textsc{Chr}(104) = $ 'h', i.e., if we see the integer 104 then this represents the character 'h'. Conversely we have that $\textsc{Ord}(\text{'h'}) = 104$. Although in a sense any consistent translation between characters and numbers would do, ASCII has some useful properties. Look specifically at the alphabetic characters:

- Imagine we want to convert a character x from lower case into upper case. The lower case characters are represented numerically as the contiguous range $97 \ldots 122$; the upper case characters as the contiguous range $65 \ldots 90$. So we can convert from lower case into upper case simply by subtracting 32. For example:

$$\textsc{Chr}(\textsc{Ord}(\text{'a'}) - 32) = \text{'A'}.$$

y	CHR(y)	y	CHR(y)	y	CHR(y)	y	CHR(y)
ORD(x)	x	ORD(x)	x	ORD(x)	x	ORD(x)	x
0	*NUL*	1	*SOH*	2	*STX*	3	*ETX*
4	*EOT*	5	*ENQ*	6	*ACK*	7	*BEL*
8	*BS*	9	*HT*	10	*LF*	11	*VT*
12	*FF*	13	*CR*	14	*SO*	15	*SI*
16	*DLE*	17	*DC*1	18	*DC*2	19	*DC*3
20	*DC*4	21	*NAK*	22	*SYN*	23	*ETB*
24	*CAN*	25	*EM*	26	*SUB*	27	*ESC*
28	*FS*	29	*GS*	30	*RS*	31	*US*
32	*SPC*	33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	*DEL*

Table 1.2 A table describing the ASCII character encoding.

- Imagine we want to test if one character x is alphabetically before some other y (ignoring the issue of case). The way the ASCII translation is specified, we can simply compare their numeric representation: if we find $\text{ORD}(x) < \text{ORD}(y)$, then the character x is before the character y in the alphabet.

The ASCII table only has 128 entries: one represents each character with a 7-bit binary vector, or an integer in the range $0 \ldots 128$. However, given that an 8-bit byte is a more convenient size, what about characters that are assigned to the range $128 \ldots 255$? Strict ASCII uses the eighth bit for error correction, but some systems ignore this and define a so-called “extended” ASCII table: characters within it are used for special purposes on a per-computer basis: one computer might give them one meaning, another computer might give them another meaning. For example, characters for specific languages are often defined in this range (e.g., é or ø), and “block” characters are included and used by artists who form text-based pictures.

The main problem with ASCII in a modern context is the lack of spare space in the encoding: it only includes characters suitable for English, so if additional language specific characters are required it cannot scale easily. Although some extensions of ASCII have been proposed, a new standard for extended character sets called **unicode** is now more prevalent.

1.4.3.3 Representing Rational Numbers

The most obvious way to represent rational numbers is simply to allocate a number of bits to the fractional part and the rest to the whole part. For example, given that we have

$$123.3125_{(10)} = 1111011.0101_{(2)},$$

we could simply have an 11-bit value $11110110101_{(2)}$ with the convention that the four least-significant bits represent the fractional part while the rest represent the whole part. This approach, where we have a fixed number of digits in the fractional part, is termed a fixed-point representation. The problem however is already clear: if we have only eight bits to store our value in, there needs to be some compromise in terms of precision. That is, since we need seven bits for the whole part, there is only one bit left for the fractional part. Hence, we either truncate the number as

$$1111011.0_{(2)} = 123.0_{(10)}$$

or perform some form of rounding. This underlines the impact of our finite resources. While n limits the range of integers that can be represented, it also impacts on the precision of rational numbers since they can only ever be approximations.

Definition 29. We say that a number x is in a base-b **fixed point** representation if written as

$$x = \pm m \cdot b^c$$

where $m, c \in \mathbb{Z}$. The value m is called the **mantissa** while the fixed value c is the **exponent**; c scales m to the correct rational value and thus determines the number of digits in the fractional part. Note that c is not included in the representation of x, it is a fixed property (or parameter) across all values.

Taking the example above, setting $b = 2$, $c = 1$ and $m = 11110110_{(2)}$ we see that

$$\begin{aligned} 11110110_{(2)} \cdot 2^{-1} &= 1111011.0_{(2)} \\ &= \quad 123.0_{(10)}. \end{aligned}$$

Not many languages have built-in types for fixed point arithmetic and there are no real standards in this area. However, despite the drawbacks, lots of applications exist where only limited precision is required, for example in storing financial or money-related values where typically only two fractional digits are required.

Floating point numbers are a more flexible approximation of rational numbers. Instead of there being a fixed number of digits in the fractional and whole parts, we let the exponent vary.

Definition 30. We say that a number x is in a base-b **floating point** representation if written as

$$x = -1^s m \cdot b^e$$

where $m, e \in \mathbb{Z}$. The value m is called the **mantissa** while e is the **exponent**; s is the **sign bit** used to determine the sign of the value.

Informally, such numbers look like

$$x = \pm \underbrace{dd \ldots dd.dd \ldots dd}_{p \text{ digits}}$$

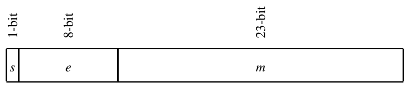
where there are p base-b digits in total. Roughly speaking, p relates to the precision of the number. We say that x is **normalised** if it is of the form

$$x = \pm \underbrace{d.dd \ldots dd}_{p \text{ digits}}$$

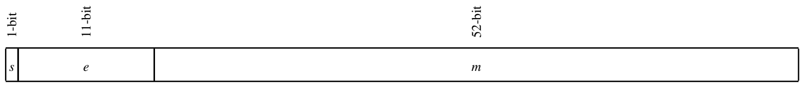
such that there is a single non-zero digit to the left of fractional point. In fact, when dealing with binary numbers if this leading digit is non-zero it *must* be one. Hence we do not need to store the leading digit and hence get one bit more of precision instead; this implicit leading digit is often called a **hidden digit**.

So that computers can inter-operate with common data formats, IEEE-754 offers two standard floating point formats which allow one to represent floating point numbers as bit-vectors. These two formats are termed single and double precision; they fit into 32-bit and 64-bit representations respectively. Both formats, described in Figure 1.3 and Listing 1.1, consist of three parts; these parts are concatenated together to form one bit-vector. A sign-magnitude method is adopted for the whole number in that a single bit denotes the sign.

The integer exponent and mantissa are both represented by bit-vectors. We need the exponent to be signed so both large and small numbers can be specified; the



(a) The 32-bit, single precision format.



(b) The 64-bit, double precision format.

Figure 1.3 Single and double precision IEEE floating point formats (as bit-fields).

```
struct ieee32
{
  uint32 m : 23, // mantissa
         e :  8, // exponent
         s :  1; // sign
}

struct ieee64
{
  uint64 m : 52, // mantissa
         e : 11, // exponent
         s :  1; // sign
}
```

Listing 1.1 Single and double precision IEEE floating point formats (as C structures).

mantissa need not be signed since we already have a sign bit for the whole number. One might imagine that a twos-complement representation of the exponent might be appropriate but the IEEE-754 standard uses an alternative approach called **biasing**. Essentially this just means that the exponent is scaled so it is always positive; for the single precision format this means it is stored with 127 added to it so the exponent 0 represents 2^{-127} while the exponent 255 represents 2^{+128}. Although this requires some extra decoding when operating with the values, roughly speaking all choices are made to make the actual arithmetic operations easier.

Again consider our example number $x = 123.3125_{(10)}$ which in binary is $x = 1111011.0101_{(2)}$. First we normalise the number to get

$$x = 1.1110110101_{(2)} \cdot 2^6$$

because now there is only one digit to the left of the binary point. Recall we do not need to store the hidden digit. So if we want a single precision, 32-bit IEEE-754 representation, our mantissa and exponent are the binary vectors

$$\begin{aligned} m &= 11101101010000000000000 \\ e &= 10000101 \end{aligned}$$

while the sign bit is 0 since we want a positive number. Using space simply to separate the three parts of the bit-vector, the representation is thus the vector

$$0 \;\; 10000101 \;\; 11101101010000000000000.$$

In any floating point representation, we need to allocate special values for certain quantities which can occur. For example, we reserve particular values to represent $+\infty$, $-\infty$ and *NaN*, or **not-a-number**. The values $+\infty$ and $-\infty$ occur when a result overflows beyond the limits of what can be represented; *NaN* occurs, for example, as a result of division by zero. For single precision, 32-bit IEEE-754 numbers these special values are

$$\begin{aligned}
0 \;\; 00000000 \;\; 00000000000000000000000 &= +0 \\
1 \;\; 00000000 \;\; 00000000000000000000000 &= -0 \\
0 \;\; 11111111 \;\; 00000000000000000000000 &= +\infty \\
1 \;\; 11111111 \;\; 00000000000000000000000 &= -\infty \\
0 \;\; 11111111 \;\; 00000100000000000000000 &= NaN \\
1 \;\; 11111111 \;\; 00100010001001010101010 &= NaN
\end{aligned}$$

with similar forms for the double precision, 64-bit format.

Finally, consider the case where the result of some arithmetic operation (or conversion) requires more digits of precision than are available. Most people learn a simple method for **rounding** in school: the value $1.24_{(10)}$ is rounded to $1.2_{(10)}$ using two digits of precision because the last digit is less than five, while $1.27_{(10)}$ is rounded to $1.3_{(10)}$ since the last digit is greater than or equal to five. Essentially, this rounds the ideal result, i.e., the result if one had infinite precision, to the most suitable representable result according to some scheme which we call the **rounding mode**. In order to compute sane results, one needs to know exactly how a given rounding mode works. IEEE-754 specification mandates the availability of four such modes; in each case, imagine we have an ideal result x. To round x using p digits of precision, one directly copies the digits $x_0 \ldots x_{p-1}$. The task of the rounding mode is then to patch the value of x_{p-1} in the right way. Throughout the following description we use decimal examples for clarity; note that essentially the same rules with minor alterations apply in binary:

Round to nearest Sometimes termed **Banker's Rounding**, this scheme alters basic rounding to provide more specific treatment when the result x lies exactly half way between two values, i.e., when $x_p = 5$ and all trailing digits from x_{p+1} onward are zero:

- If $x_p \leq 4$, then take x_{p-1} and do not alter it.
- If $x_p \geq 6$, then take x_{p-1} and alter it by adding one.
- If $x_p = 5$ and at least one of the digits x_{p+1} onward is non-zero, then take x_{p-1} and alter it by adding one.

- If $x_p = 5$ and all of the digits x_{p+1} onward are zero, then alter x_{p-1} to the nearest even digit. That is:
  - If $x_{p-1} \equiv 0 \pmod 2$, then do not alter it.
  - If $x_{p-1} \equiv 1 \pmod 2$, then alter it by adding one.

Again considering the case of rounding for two digits of precision we find that $1.24_{(10)}$ is rounded to $1.2_{(10)}$ since the p-th digit is less than than or equal to four; $1.27_{(10)}$ is rounded to $1.3_{(10)}$ since the p-th digit is greater than or equal to six; $1.251_{(10)}$ is rounded to $1.3_{(10)}$ since the p-th digit is equal to five and the $(p+1)$-th digit is non-zero; $1.250_{(10)}$ is rounded to $1.2_{(10)}$ since the p-th digit is equal to five, digit $(p+1)$ and onward are zero, and the $(p-1)$-th digit is even; and $1.350_{(10)}$ is rounded to $1.4_{(10)}$ since the p-th digit is equal to five, digit $(p+1)$ and onward are zero, and the $(p-1)$-th digit is odd.

Round toward $+\infty$ This scheme is sometimes termed **ceiling**. In the case of a positive number, if the p-th digit is non-zero the $(p-1)$-th digit is altered by adding one. In the case of a negative number, the p-th digit onward is discarded. For example $1.24_{(10)}$ and $1.27_{(10)}$ would both round to $1.3_{(10)}$ while $1.20_{(10)}$ rounds to $1.2_{(10)}$; $-1.24_{(10)}$, $-1.27_{(10)}$ and $-1.20_{(10)}$ all round to $-1.2_{(10)}$.

Round toward $-\infty$ Sometimes termed **floor**, this scheme is the complement of the round toward $+\infty$ above. That is, in the case of a positive number the p-th digit onward is discarded. In the case of a negative number, if the p-th digit is non-zero the $(p-1)$-th digit is altered by adding one. For example, $-1.24_{(10)}$ and $-1.27_{(10)}$ would both round to $-1.3_{(10)}$ while $-1.20_{(10)}$ rounds to $-1.2_{(10)}$; $1.24_{(10)}$, $1.27_{(10)}$ and $1.20_{(10)}$ all round to $1.2_{(10)}$.

Round toward zero This scheme operates as round toward $-\infty$ for positive numbers and as round toward $+\infty$ for negative numbers. For example $1.27_{(10)}$, $1.24_{(10)}$ and $1.20_{(10)}$ round to $1.2_{(10)}$; $-1.27_{(10)}$, $-1.24_{(10)}$ and $-1.20_{(10)}$ round to $-1.2_{(10)}$.

The C standard library offers access to these features. For example the `rint` function rounds a floating point value using the currently selected IEEE-754 rounding mode; this mode can be inspected using the `fegetround` function and set using the `fesetround` function. The latter two functions use the constant values `FE_TONEAREST`, which corresponds to the round to nearest mode; `FE_UPWARD`, which corresponds to the round toward $+\infty$ mode; `FE_DOWNWARD`, which corresponds to the round toward $-\infty$ mode; and `FE_TOWARDZERO`, which corresponds to the round toward zero mode.

Using a slightly cryptic program, we can demonstrate in a very practical way that floating point representations in C work as expected. Listing 1.2 describes a C union called `ieee32` which shares space between a single precision `float` value called `f` and a 32-bit `int` called `i`; since they share the same physical space, essentially this allows us to fiddle with bits in `i` so as to provoke changes in `f`. The program demonstrates this; compiling and executing the program gives the following output:

```
bash$ gcc -std=gnu99 -o F F.c
bash$ ./F
00000000
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fenv.h>

typedef union __ieee32
{
  float  f;
  uint32 i;
}
ieee32;

int main( int argc, char* argv[] )
{
  ieee32 t;

  t.f = 2.8;

  printf( "%08X\n", ( t.i & 0x80000000 ) >> 31 );
  printf( "%08X\n", ( t.i & 0x7F800000 ) >> 23 );
  printf( "%08X\n", ( t.i & 0x007FFFFF ) >>  0 );

  t.i |= 0x80000000;
  printf( "%f\n", t.f );

  t.i &= 0x807FFFFF;
  t.i |= 0x40800000;
  printf( "%f\n", t.f );

  t.i  = 0x7F800000;
  printf( "%f\n", t.f );

  t.i  = 0x7F808000;
  printf( "%f\n", t.f );
}
```

Listing 1.2 A short C program that performs direct manipulation of IEEE floating point numbers.

```
00000080
00333333
-2.800000
-5.600000
inf
nan
```

The question is, what on earth does this mean ? The program first constructs an instance of the `ieee32` union called `t` and sets the floating point field to $2.8_{(10)}$. The `printf` statements that follow extract and print the sign, exponent and mantissa fields of `t.f` by examining the corresponding bits in `t.i`. The output shows that the sign field is $0_{(16)}$ (i.e., the number is positive), the exponent field is $80_{(16)}$ (i.e., after considering the bias, the exponent is $1_{(16)}$) and the mantissa field is $333333_{(16)}$. If we write this as a bit-vector as above, we have

$$0 \ \ 10000000 \ \ 01100110011001100110011$$

which is representing the actual value

$$-1^0 \cdot 1.01100110011001100110011001 1_{(2)} \cdot 2^1$$

or $2.8_{(10)}$ as expected. Next, the program ORs the value $80000000_{(16)}$ with `t.i` which has the effect of setting the MSB to one; since the MSB of `t.i` mirrors the sign of `t.f`, we expect this to have negated `t.f`. As expected, the next `printf` produces $-2.8_{(10)}$ to verify this. The program then ANDs `t.i` with $807FFFFF_{(16)}$ and then ORs it with $40800000_{(16)}$. The net effect is to set the exponent field of `t.f` to the value $129_{(10)}$ rather than $128_{(10)}$ as before, or more simply to multiply the value by two. Again as expected, the next `printf` produces $-5.6_{(10)} = 2 \cdot -2.8_{(10)}$ to verify this. Finally, the program sets `t.i` to the representations we would expect for $+\infty$ and *NaN* to demonstrate the output in these cases.

1.5 Toward a Digital Logic

We have already hinted that digital computers are happiest dealing with the binary values zero and one since they can be easily mapped onto low and high electrical signals. We have also seen that it is possible to define a Boolean algebra over the set $\mathbb{B} = \{0,1\}$ where the meaning of the operators $\wedge$, $\vee$ and $\neg$ is described by the following table:

x y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$
0 0	1	0	0	0
0 1	1	0	1	1
1 0	0	0	1	1
1 1	0	1	1	0

Note that we have included $x \oplus y$ to denote exclusive-or, which we call XOR. In a sense, this is just a shorthand since

$$x \oplus y = (\neg x \wedge y) \vee (x \wedge \neg y)$$

such that XOR is defined in terms of AND, OR and NOT. Now consider all the pieces of theory we have accumulated so far:

- We know that the operators AND, OR and NOT obey the axioms outlined previously and can therefore construct Boolean expressions and manipulate them while preserving their meaning.
- We can describe Boolean formula of the form

  $$\mathbb{B}^n \to \mathbb{B}$$

  and hence also construct functions of the form

  $$\mathbb{B}^n \to \mathbb{B}^m$$

simply by having m functions and grouping their outputs together. It thus makes sense that NOT, AND, OR and XOR can be well-defined for the set $\mathbb{B}^n$ as well as $\mathbb{B}$. We just define n Boolean functions such that, for example,

$$f_i : \mathbb{B} \rightarrow \mathbb{B}, \ \ f_i(x,y) = x_i \wedge y_i.$$

So for $x,y,z \in \{0,1\}^n$ we can write $z_i = f_i(x,y)$ with $0 \leq i < n$ to perform the AND operation component-wise on all the bits from x and y. In fact, we do not bother with this long-winded approach and just write $z = x \wedge y$ to mean the same thing.

- We know how to represent numbers as members of the set $\mathbb{B}^n$ and have described how bit-vectors are just a short hand way of writing down such representations. Since we can build arbitrary Boolean functions of the form

$$\mathbb{B}^n \rightarrow \mathbb{B}^m,$$

the results of Shannon which show how such functions can perform arithmetic on our number representations should not be surprising. For example, imagine we want to add two integers together. That is, say $x,y,z \in \mathbb{Z}$ are represented as n-bit vectors $p,q,r \in \mathbb{B}^n$. To calculate $z = x + y$, all we need to do is make n similar Boolean functions, say f_i, such that $r_i = f_i(p,q)$. Each function takes p and q, the representation of x and y, as input and produces a single bit of r, the representation of z, as output.

In short, by investigating how Boolean algebra unifies propositional logic and set theory and how to represent numbers using sets, we have a system which is ideal for representing and manipulating such numbers using digital components. At the most basic level this is all a computer does: we have formed a crucial link between what seems like a theoretical model of computation and the first step toward a practical realisation.

1.6 Further Reading

- D. Goldberg.
  What Every Computer Scientist Should Know About Floating-Point Arithmetic.
  In *ACM Computing Surveys*, **23**(1), 5–48, 1991.
- N. Nissanke.
  Introductory Logic and Sets for Computer Science.
  Addison-Wesley, 1998. ISBN: 0-201-17957-1.
- C. Petzold.
  Code: Hidden Language of Computer Hardware and Software.
  Microsoft Press, 2000. ISBN: 0-735-61131-9.

- K.H. Rosen.
  Discrete Mathematics and it's Applications.
  McGraw-Hill, 2006. ISBN: 0-071-24474-3.
- J. Truss.
  Discrete Mathematics for Computer Scientists.
  Addison-Wesley, 1998. ISBN: 0-201-36061-6.

1.7 Example Questions

1. For the sets $A = \{1,2,3\}$, $B = \{3,4,5\}$ and $\mathscr{U} = \{1,2,3,4,5,6,7,8\}$ compute the following:

a. $|A|$.
b. $A \cup B$.
c. $A \cap B$.
d. $A - B$.
e. $\overline{A}$.
f. $\{x : 2 \cdot x \in \mathscr{U}\}$.

2. For each of the following decimal integers, write down the 8-bit binary representation in sign-magnitude **and** twos-complement:

a. $+0$.
b. -0.
c. $+72$.
d. -34.
e. -8.
f. 240.

3. For some 32-bit integer x, explain what is meant by the Hamming weight of x; write a short C function to compute the Hamming weight of an input value.

4. Show that
$$(\neg x \wedge y) \vee (\neg y \wedge x) \vee (\neg x \wedge \neg y) = \neg x \vee \neg y.$$