2

Mathematical Preliminaries

In this chapter, we introduce many of the fundamental mathematical ideas used throughout the book. We first discuss *sets*, which help organize "things" into meaningful unordered collections. We then discuss functions which help map things to other things. Next, we discuss relations that relate things. We provide a concrete syntax, that of *Lambda expressions*, for writing down function definitions. We then present ways to "count" infinite sets through a measure known as *cardinality*.

2.1 Numbers

We will refer to various classes of numbers. The set Nat, or *natural numbers*, refers to whole numbers greater than or equal to zero, i.e., $0, 1, 2, \ldots$. The set Int, or *integers*, refers to whole numbers, i.e., $0, 1, -1, 2, -2, 3, -3 \ldots$. The set $Real$, or *real numbers*, refers to both rational as well as irrational numbers, including 0.123, $\sqrt{2}$, π, 1, and -2.

2.2 Boolean Concepts, Propositions, and Quantifiers

We assume that the reader is familiar with basic concepts from Boolean algebra, such as the use of the Boolean connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). We will be employing the two *quantifiers* "for all" ($\forall$) and "for some" or equivalently "there exists" ($\exists$) in many definitions. Here we provide preliminary discussions about these operators; more details are provided in Section 5.2.4.

In a nutshell, the quantifiers $\forall$ and $\exists$ are iteration schemes for $\wedge$, $\vee$, and $\neg$, much like Σ (summation) and Π (product) are iteration schemes for *addition* ($+$) and *multiplication* ($\times$). The universal quantification operator $\forall$ is used to make an assertion about all the objects in the domain of discourse. (In mathematical logic, these domains are assumed to be non-empty). Hence,

$$\forall x : Nat : \ P(x)$$

is equivalent to an infinite conjunction

$$P(0) \ \wedge \ P(1) \ \wedge \ P(2) \ \ldots$$

or equivalently $\wedge_{x \in Nat} : \ P(x)$.

2.3 Sets

A *set* is a collection of things. For example, $A = \{1, 2, 3\}$ is a set containing three natural numbers. The order in which we list the contents of a set does not matter. For example, $A = \{3, 1, 2\}$ is the same set as above. A set cannot have duplicate elements. For example, $B = \{3, 1, 2, 1\}$ is not a set.[1]

A set containing no elements at all is called the *empty set*, written $\{\}$, or **equivalently**, $\emptyset$. A set may also consist of a collection of other sets, as in

$$P = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

P has a special status; it contains every *subset* of set A. P is in fact the *powerset* of A. We will have more to say about powersets soon.

2.3.1 Defining sets

Sets are specified using the *set comprehension* notation

$$S = \{x \in D \ \mid \ p(x)\}.$$

Here, S includes all x from some universe D such that $p(x)$ is true. $p(x)$ is a Boolean formula called *characteristic formula*. p by itself is called the *characteristic predicate*. We can leave out D if it is clear from the context.

[1] An unordered collection with duplicates, such as B, is called a *multi-set* or *bag*.

Examples:

- Set A, described earlier, can be written as

$$A = \{x \mid x = 1 \vee x = 2 \vee x = 3\}.$$

- For any set D,

$$\{x \in D \mid true\} = D.$$

  Notice from this example that the characteristic formula can simply be $true$, or for that matter $false$.
- For any set D,

$$\{x \in D \mid false\} = \emptyset.$$

The next two sections illustrate that care must be exercised in writing set definitions. The brief message is that by writing down a collection of mathematical symbols, one does not necessarily obtain something that is well defined. Sometimes, we end up defining more than one thing without realizing it (the definitions admit multiple solutions), and in other cases we may end up creating contradictions.

2.3.2 Avoid contradictions

Our first example illustrates the famous *Russell's Paradox*. This paradox stems from allowing expressions such as $x \in x$ and $x \notin x$ inside characteristic formulas. Consider some arbitrary domain D. Define a set S as follows:

$$S = \{x \in D \mid x \notin x\}.$$

Now, the expression $x \notin x$ reveals that x itself is a set. Since S is a set, we can now ask, "is S a member of S?"

- If S is a member of S, it cannot be in S, because S cannot contain sets that contain themselves.
- However, if S is not a member of S, then S must contain S!

Contradictions are required to be complete, *i.e.*, apply to all possible cases. For example, if $S \notin S$ does not result in a contradiction, that, then, becomes a consistent solution. In this example, we fortunately obtain a contradiction in all the cases. The proposition $S \in S$ must produce a definite answer - true or false. However, *both* answers lead to a contradiction.

We can better understand this contradiction as follows. For Boolean quantities a and b, let $a \Rightarrow b$ stand for "a implies b" or "if a then b;" in other words, $\Rightarrow$ is the *implication* operator. Suppose $S \in S$. This

allows us to conclude that $S \notin S$. In other words, $(S \in S) \Rightarrow (S \notin S)$ is true. In other words, $\neg(S \in S) \vee (S \notin S)$, or $(S \notin S) \vee (S \notin S)$, or $(S \notin S)$ is true. Likewise, $(S \in S) \Rightarrow (S \notin S)$. This allows us to prove $(S \in S)$ true. Since we have proved $S \in S$ as well as $S \notin S$, we have proved their conjunction, which is $false$! With $false$ proved, anything else can be proved (since $false \Rightarrow anything$ is $\neg(false) \vee anything$, or $true$). Therefore, it is essential to avoid contradictions in mathematics.

Russell's Paradox is used to conclude that a "truly universal set" – a set that contains *everything* – cannot exist. Here is how such a conclusion is drawn. Notice that set S, above, was defined in terms of an *arbitrary* set called D. Now, if D were to be a set that contains "everything," a set such as S must clearly be present inside D. However, we just argued that S must not exist, or else a contradiction will result. Consequently, a set containing *everything* cannot exist, for it will lack at least S. This is the reason why the notion of a *universal* set is not an absolute notion. Rather, a universal set specific to the domain of discourse is defined each time. This is illustrated below in the section devoted to universal sets. In practice, we disallow sets such as S by banning expressions of the form $x \in x$. In general, such restrictions are handled using *type theory* [48].

2.3.3 Ensuring uniqueness of definitions

When a set is defined, it must be *uniquely* defined. In other words, we cannot have a definition that does not pin down the exact set being talked about. To illustrate this, consider the "definition" of a set

$$S = \{x \in D \mid x \in S\},$$

where D is some domain of elements. In this example, the set being defined depends on itself. The circularity, in this case, leads to S not being uniquely defined. For example, if we select $D = Nat$, and plug in $S = \{1, 2\}$ on both sides of the equation, the equation is satisfied. However, it is also satisfied for $S = \emptyset$, $S = \{3, 4, 5\}$. Hence, in the above circular definitions, we cannot *pin down* exactly what S is.

The message here is that one must avoid using purely circular definitions. However, sets are allowed to be defined through *recursion* which, at first glance, is "a sensible way to write down circular definitions." Chapter 7 explains how recursion is understood, and how sets can be uniquely defined even though "recursion seems like circular definition."

Operations on Sets

Sets support the usual operations such as membership, *union*, *intersection*, *subset*, *powerset*, *Cartesian product*, and *complementation*. $x \in A$ means x is a member of A. The *union* of two sets A and B, written $A \cup B$, is a set such that $x \in (A \cup B)$ *if and only if* $x \in A$ *or* $x \in B$. In other words, $x \in (A \cup B)$ implies that $x \in A$ or $x \in B$. Also, $x \in A$ or $x \in B$ implies that $x \in (A \cup B)$. similarly, the *intersection* of two sets A and B, written $A \cap B$, is a set such that $x \in (A \cap B)$ if and only if $x \in A$ *and* $x \in B$.

A *proper* subset A of B, written $A \subset B$, is a subset of B different from B. $A \subseteq B$, read 'A is a subset of B', means that $A \subset B$ or $A = B$. Note that the empty set has no proper subset.

Subtraction, Universe, Complementation, Symmetric Difference

Given two sets A and B, set subtraction, '\', is defined as follows:

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}.$$

Set subtraction basically removes all the elements in A that are in B. For example, $\{1, 2\} \setminus \{2, 3\}$ is the set $\{1\}$. 1 survives set subtraction because it is not present in the second set. The fact that 3 is present in the second set is immaterial, as it is not present in the first set.

For each type of set, there is a set that contains all the elements of that type. Such a set is called the *universal* set. For example, consider the set of *all strings* over some alphabet, such as $\{a, b\}$. This is universal set, as far as sets of strings are concerned. We can write this set as

$$\text{SigmaStar} = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}.$$

(The reason why we name the above set SigmaStar will be explained in Chapter 7.) Here, ε is the empty string, commonly written as `""`, a and b are strings of length 1, aa, ab, ba, and bb are strings of length 2, and so on. While discussing natural numbers, we can regard $Nat = \{0, 1, 2, \ldots\}$ as the universe.

> The symbol ε is known to confuse many students. Think of it as the "zero" element of strings, or simply read it as the empty string `""`. By way of analogy, the analog of the arithmetic expression $0 + 1$, which simplifies to 1, is ε concatenated with a, which simplifies to a. (We express string concatenation through juxtaposition). Similarly, $0 + 2 + 0 = 2$ is to numbers as $\varepsilon\ aa\ \varepsilon = aa$ is to strings. More discussions are provided in Section 7.2.4.

Universal sets help define the notion of *complement* of a set. Consider the universal set (or "universe") SigmaStar of strings over some alphabet. The complement of a set of strings such as $\{a, ba\}$ is SigmaStar $\setminus \{a, ba\}$. If we now change the alphabet to, say, $\{a\}$, the universal set of strings over this alphabet is

$$\text{SigmaStar1} = \{\varepsilon, a, aa, aaa, aaaa, \ldots\}.$$

Taking the complement of a set such as $\{a, aaa\}$ with respect to SigmaStar1 yields a set that contains strings of a's such that the number of occurrences of a's is neither 1 nor 3.

Given two sets A and B, their *symmetric difference* is defined to be

$$(A \setminus B) \cup (B \setminus A).$$

For example, if $A = \{1, 2, 3\}$ and $B = \{2, 3, 4, 5\}$, their symmetric difference is the set $\{1, 4, 5\}$. The symmetric difference of two sets produces, in effect, the XOR (exclusive-OR) of the sets.

For any alphabet Σ and its corresponding universal set SigmaStar, the complement of the empty set $\emptyset$ is SigmaStar. One can think of $\emptyset$ as *the* empty set with respect to *every* alphabet.

Types versus Sets

The word *type* will be used to denote a *set* together with its associated operations. For example, the type *natural number*, or Nat, is associated with the set $\{0, 1, 2, \ldots\}$ and operations such as `successor`, `+`, etc. $\emptyset$ is an overloaded symbol, denoting the empty set of every type. When we use the word "type," most commonly we will be referring to the underlying set, although strictly speaking, types are "sets plus their operations."

Numbers as Sets

In mathematics, it is customary to regard natural numbers themselves as sets. Each natural number essentially denotes the set of natural numbers below it. For example, 0 is represented by $\{\}$, or $\emptyset$, as there are no natural numbers below 0. 1 is represented by $\{0\}$, or (more graphically) $\{\{\}\}$, the only natural number below 1. Similarly, 2 is the set $\{0, 1\}$, 3 is the set $\{0, 1, 2\}$, and so on. This convention comes in quite handy in making formulas more readable, by avoiding usages such as

$$\forall i \ : \ 0 \leq i \leq N - 1 \ : \ ..something..$$

and replacing them with

$$\forall i \in N \ : \ ..something..$$

Notice that this convention of viewing sets as natural numbers is exactly similar to how numbers are defined in set theory textbooks, e.g., [50]. We are using this convention simply as a labor-saving device while writing down definitions. We do not have to fear that we are suddenly allowing sets that contain other sets.

As an interesting diversion, let us turn our attention back to the discussion on Russell's Paradox discussed in Section 2.3.2. Let us take D to be the set of natural numbers. Now, the assertion $x \notin x$ evaluates to *true* for every $x \in D$. This is because no natural number (viewed as a set) contains itself - it only contains all natural numbers strictly *below* it in value. Hence, no contradiction results, and S ends up being equal to Nat.

2.4 Cartesian Product and Powerset

The *Cartesian product* operation '$\times$' helps form *sets of tuples* of elements over various types. The terminology here goes as follows: 'pairs' are 'two-tuples,' 'triples' are 'three-tuples,' 'quadruples' are 'four-tuples,' and so on. After 5 or so, you are allowed to say 'n-ple' - for instance, '37-ple' and so on. For example, **the set** $Int \times Int$ denotes the sets of pairs of all integers. Mathematically, the former set is

$$Int \times Int = \{\langle x, y\rangle \ \mid \ x \in Int \ \wedge y \in Int\}.$$

Given two sets A and B, the **Cartesian product** of A and B, written $A \times B$, is defined to be the set

$$\{\langle a, b\rangle \ \mid \ a \in A \ \wedge \ b \in B\}.$$

We can take Cartesian product of multiple sets also. In general, the Cartesian product of n sets $A_i, \ i \in n$ results in a set of "n-tuples"

$$A_0 \times A_1 \times \ldots A_{n-1} = \{\langle a_0, a_1, \ldots, a_{n-1}\rangle \ \mid \ a_i \in A_i \ for \ every \ i\}.$$

If one of these sets, A_i, is $\emptyset$, the Cartesian product results in $\emptyset$ because it is impossible to "draw" any element out of A_i in forming the n-ples. Here are some examples of Cartesian products:

- $\{2, 4, 8\} \times \{1, 3\} \times \{100\} =$
  $\{\langle 2, 1, 100\rangle, \langle 2, 3, 100\rangle, \langle 4, 1, 100\rangle, \langle 4, 3, 100\rangle, \langle 8, 1, 100\rangle, \langle 8, 3, 100\rangle\}$.

- SigmaStar1×SigmaStar1 = $\{\langle x, y\rangle \mid x \text{ and } y \text{ are strings over } \{a\}\}$.

In taking the Cartesian product of n sets A_i, $i \in n$, it is clear that if $n = 1$, we get the set A_0 back. For example, if $A_0 = \{1, 2, 3\}$, the 1-ary Cartesian product of A_0 is itself. Note that $A_0 = \{1, 2, 3\}$ can also be written as $A_0 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$ because, in classical set theory [50], 1-tuples such as $\langle 0\rangle$ are the same as the item without the tuple sign (in this case 0).

It is quite common to take the Cartesian product of different *types* of sets. For example, the set $Int \times Bool$ denotes the sets of pairs of integers and Booleans. An element of the above set is $\langle 22, true\rangle$, which is a pair consisting of one integer and one Boolean.

2.4.1 Powersets and characteristic sequences

The **powerset** of a set S is the set of all its subsets. As is traditional, we write 2^S to denote the powerset of S. In symbols,

$$2^S = \{x \mid x \subseteq S\}.$$

This "exponential" notation suggests that the size of the powerset is 2 raised to the size of S. We can argue this to be the case using the notion of *characteristic sequences*. Take $S = \{1, 2, 3\}$ for example. Each subset of S is defined by a bit vector of length three. For instance, 000 represents $\emptyset$ (include none of the elements of S), 001 represents $\{3\}$, 101 represents $\{1, 3\}$, and 111 represents S. These "bit vectors" are called *characteristic sequences*. All characteristic sequences for a set S are of the same length, equal to the size of the set, $|S|$. Hence, the number of characteristic sequences for a *finite* set S is exponential in $|S|$.

2.5 Functions and Signature

A function is a mathematical object that expresses how items called "inputs" can be turned into other items called "outputs." A function maps its *domain* to its *range*; and hence, the inputs of a function belong to its *domain* and the outputs belong to its *range*. **The domain and range of a function are always assumed to be non-empty.** The expression "$f : T_D \to T_R$" is called the signature of f, denoting that f maps the domain of type T_D to the range of type T_R. Writing signatures down for functions makes it very clear as to what the function "inputs" and what it "outputs." Hence, this is a highly recommended practice.

As a simple example, $+ : Int \times Int \rightarrow Int$ denotes the signature of integer addition.

Function signatures must attempt to capture their domains and ranges as tightly as possible. Suppose we have a function g that accepts subsets of $\{1, 2, 3\}$, outputs 4 if given $\{1, 2\}$, and outputs 5 given anything else. How do we write the signature for g? Theoretically speaking, it is correct to write the signature as $2^{Nat} \rightarrow Nat$; however, in order to provide maximum insight to the reader, one must write the signature as

$$2^{\{1,2,3\}} \rightarrow \{4, 5\}.$$

If you are unsure of the exact domain and range, try to get as tight as possible. Remember, you must help the reader.
The *image* of a function is the set of range points that a function actually maps onto. For function $f : T_D \rightarrow T_R$,

$$image(f) = \{y \in T_R \mid \exists x \in T_D \ : \ y = f(x)\}.$$

2.6 The λ Notation

The Lambda calculus was invented by Alonzo Church[2] as a formal representation of computations. Church's thesis tells us that the lambda-based evaluation machinery, Turing machines, as well as other formal models of computation (Post systems, Thue systems, ...) are all formally equivalent. Formal equivalences between these systems have all been worked out by the 1950s.

More immediately for the task at hand, the lambda notation provides a *literal* syntax for *naming* functions. First, let us see in the context of numbers how we name them. The sequence of numerals (in programming parlance, the *literal*) '`1`' '`9`' '`8`' '`4`' names the number 1984. We do not need to give alternate names, say 'Fred', to such numbers! A numeral sequence such as `1984` suffices. In contrast, during programming one ends up giving such alternate names, typically derived from the domain of discourse. For example,

function `Fred`(x) {return 2;}.

Using Lambda expressions, one can write such function definitions without using alternate names. Specifically, the Lambda expression $\lambda x.2$ captures the same information as in the above function definition.

[2] When once asked how he chose λ as the delimiter, Church replied, "Eenie meenie mynie mo!"

We think of strings such as $\lambda x.2$ as a *literal* (or *name*) that describes functions. In the same vein, using Lambda calculus, one name for the successor function is $\lambda x.(x+1)$, another name is $\lambda x.(x+2-1)$, and so on.[3] We think of *Lambda expressions* as *irredundant* names for functions (irredundant because redundant strings such as 'Fred' are not stuck inside them). We have, in effect, "de-Freded" the definition of function Fred. In Chapter 6, we show that this idea of employing irredundant names works even in the context of *recursive* functions. While it may appear that performing such 'de-Fredings' on recursive definitions appears nearly impossible, Chapter 6 will introduce a trick to do so using the so-called `Y` operator. Here are the only two rules pertaining to it that you need to know:

- *The Alpha rule*, or "the name of the variable does not matter." For example, $\lambda x.(x+1)$ is the same as $\lambda y.(y+1)$. This process of renaming variables is called *alpha conversion*. Plainly spoken, the Alpha rule simply says that, in theory,[4] the formal parameters of a function can be named however one likes.
- *The Beta rule*, or "here is how to perform a function call." A function is applied to its argument by writing the function name and the argument name in juxtaposition. For example, $(\lambda x.(x+1))\ 2$ says "feed" 2 in place of x. The result is obtained by substituting 2 for x in the body $(x+1)$. In this example, $2+1$, or 3 results. This process of simplification is called *beta reduction*.

The formal arguments of Lambda expressions associate to the right. For example, as an abbreviation, we allow cascaded formal arguments of the form $(\lambda xy.(x+y))$, as opposed to writing it in a fully parenthesized manner as in $(\lambda x.(\lambda y.(x+y)))$. In addition, the arguments to a Lambda expression associate to the left. Given these conventions, we can now illustrate the simplification of Lambda expressions. In particular,

$$(\lambda zy.(\lambda x.(z+x)))\ 2\ 3\ 4$$

can be simplified as follows (we show the bindings introduced during reduction explicitly):

$$= (\lambda zy.(\lambda x.(z+x)))\ 2\ 3\ 4$$

[3] You may be baffled that I suddenly use "23" and "+" as if they were Lambda terms. As advanced books on Lambda calculus show [48], such quantities can also be encoded as Lambda expressions. Hence, anything that is *effectively computable*—computable by a machine—can be formally defined using only the Lambda calculus.

[4] In practice, one chooses mnemonic names.

$= $ (using the Beta rule) $(\lambda \underline{z = 2}\ \underline{y = 3}.(\lambda x.(z + x)))4.$
$= (\lambda x.(2 + x))4$
$= $ (using the Beta rule) $(\lambda \underline{x = 4}.(2 + x))$
$= 2 + 4$
$= 6$

The following additional examples shed further light on Lambda calculus:

- $(\lambda x.x)\ 2$ says apply the identity function to argument 2, yielding 2.
- $(\lambda x.x)\ (\lambda x.x)$ says "feed the identity function to itself." Before performing beta reductions here, we are well-advised to perform alpha conversions to avoid confusion. Therefore, we turn $(\lambda x.x)\ (\lambda x.x)$ into $(\lambda x.x)\ (\lambda y.y)$ and then apply beta reduction to obtain $(\lambda y.y)$, or the identity function back.
- As the Lambda calculus seen so far does not enforce any "type checking," one can even feed $(\lambda x.(x + 1))$ to itself, obtaining (after an alpha conversion) $(\lambda x.(x+1))+1$. Usually such evaluations then get "stuck," as we cannot add a number to a function.

2.7 Total, Partial, 1-1, and Onto Functions

Functions that are defined over their entire domain are *total*. An example of a total function is $\lambda x.2x$, where $x \in Nat$. A *partial* function is one undefined for some domain points. For example, $\lambda x.(2/x)$ is a partial function, as it is undefined for $x = 0$.

The most common use of partial functions in computer science is to model *programs that may go into infinite loops for some of their input values*. For example, the recursive program over Nat,

$$f(x) = if\ (x = 0)\ then\ 1\ else\ f(x)$$

terminates only for $x = 0$, and loops for all other values of x. Viewed as a function, it maps the domain point 0 to the range point 1, and is undefined everywhere else on its domain. Hence, function f can be naturally modeled using a *partial function.* In the Lambda notation, we can write f as $\lambda x.if\ (x = 0)\ then\ 1$. Notice that we use an "if-then" which leaves the "else" case undefined.

One-to-one (1-1) functions f are those for which every point $y \in image(f)$ is associated with exactly one point x in T_D. A function that is not 1-1 is 'many-to-one.' One-to-one functions are also known as *injections*. An example of an injection is the predecessor function $pred : Nat \rightarrow Nat$, defined as follows:

$$\lambda x.if\ (x > 0)\ then\ (x - 1).$$

We have $pred(1) = 0$, $pred(2) = 1$, and so on. This function is partial because it is undefined for 0.

Onto functions f are those for which $image(f) = T_R$. Onto functions are also known as *surjections*. While talking about the type of the range, we say function f maps *into* its range-type. Hence, *onto* is a special case of *into* when the entire range is covered. **One-to-one, onto, and total functions are known as bijections**. Bijections are also known as **correspondences**.

Examples: We now provide some examples of various types of functions. In all these discussions, assume that $f : Nat \rightarrow Nat$.

- An example of a one-to-one (1-1) function ("injection") is $f = \lambda x.2x$.
- An example of a *many-to-one* function is $f = \lambda x.(x\ mod\ 4)$.
- An example of an onto function is $f = \lambda x.x$.
- An example of a partial function is $f = \lambda x.if\ even(x)\ then\ (x/2)$.
- An example of a bijection is $f = \lambda x.if\ even(x)\ then\ (x{+}1)\ else\ (x{-}1)$. All bijections $f : T_D \rightarrow T_R$ where $T_D = T_R = T$ are the same type, are *permutations* over T.
- Into means *not necessarily onto.* A special case of *into* is *onto.*
- Partial means *not necessarily total.* A special case of *partial* is *total.*

For any given one-to-one function f, we can define its *inverse* to be f^{-1}. This function f^{-1} is defined at all its *image* points. Therefore, whenever f is defined at x,

$$f^{-1}(f(x)) = x.$$

For $f : T_D \rightarrow T_R$, we have $f^{-1} : T_R \rightarrow T_D$. Consequently, if f is *onto*, then f^{-1} is *total*—defined everywhere over T_R. To illustrate this, consider the predecessor function, *pred*. The image of this function is Nat. Hence, *pred* is onto. Hence, while $pred : Nat \rightarrow Nat$ is not total, $pred^{-1} : Nat \rightarrow Nat$ is total, and turns out to be the *successor* function *succ*.

Given a bijection f with signature $T_D \rightarrow T_R$, for any $x \in T_D$, $f^{-1}(f(x)) = x$, and for any $y \in T_R$, $f(f^{-1}(y)) = y$. This shows that if f is a bijection from T_D to T_R, f^{-1} is a bijection from T_R to T_D. For this reason, we tend to call f a bijection **between** T_D and T_R - given the forward mapping f, the existence of the backward mapping f^{-1} is immediately guaranteed.

Composition of functions

The composition of two functions $f : A \to B$ and $g : B \to C$, written $g \circ f$, is the function $\lambda x \,.\, g(f(x))$.

2.8 Computable Functions

Computational processes map their inputs to their outputs, and therefore are naturally modeled using functions. For instance, given two matrices, a computational process for matrix multiplication yields the product matrix. All those functions whose mappings may be obtained through a *mechanical process* are called *computable* functions, *effectively computable* functions, or *algorithmically computable* functions. For practical purposes, another equivalent definition of a computable function is one whose definition can be expressed in a general-purpose programming language. By 'mechanical process,' we mean a sequence of elementary steps, such as bit manipulations, that can be carried out on a machine. Such a process must be finitary, in the sense that for any input for which the function is defined, the computational process producing the mapping must be able to read the input in finite time and yield the output in a *finite* amount of time. Chapter 3 discusses the notion of a 'machine' a bit more in detail; for now, think of computers when we refer to a machine.

Non-computable functions are well-defined mathematical concepts. These are genuine mathematical functions, albeit those whose mappings cannot be obtained using a machine. In Section 3.1, based on cardinality arguments, we shall show that non-computable functions do exist. We hope that the intuitions we have provided above will allow you to answer the following problems intuitively. The main point we are making in this section is that just because a function "makes sense" mathematically doesn't necessarily mean that we can code it up as a computer program!

2.9 Algorithm versus Procedure

An *algorithm* is an *effective procedure*, where the word 'effective' means 'can be broken down into elementary steps that can be carried out on a computer.' The term *algorithm* is reserved to those procedures that come with a guarantee of termination on every input. If such a guarantee is not provided, we must not use the word 'algorithm,' but instead use the word *procedure*. While this is a simple criterion one

can often apply, sometimes it may not be possible to tell whether to call something an algorithm or a procedure. Consider the celebrated "$3x+1$ problem," also known as "Collatz's problem," captured by the following program:

```
function three_x_plus_one(x)
 { if (x==1) then return 1;
   if even(x) then three_x_plus_one(x/2);
   else three_x_plus_one(3x+1); }
```

For example, given `3`, the `three_x_plus_one` function obtains 10, 5, 16, 8, 4, 2, 1, and halts. Will this function halt for all `x`? Nobody knows! It is still open whether this function will halt for all `x` in Nat [24]! Consequently, if someone were to claim that the above program is their actual implementation of an *algorithm* (not merely a *procedure*) to realize the constant function $\lambda x.1$, not even the best mathematicians or computer scientists living today would know how to either confirm or to refute the claim! That is, nobody today is able to prove or disprove that the above program will halt for all x, yielding 1 as the answer.[5]

2.10 Relations

Let S be a set of k-tuples. Then a k-ary relation R over S is defined to be a subset of S. It is also quite common to assume that the word 'relation' means 'binary relation' ($k = 2$); we will not follow this convention, and shall be explicit about the arity of relations. For example, given $S = Nat \times Nat$, we define the binary relation $<$ over S to be

$$< \; = \{\langle x, y\rangle \;\mid\; x, y \in Nat \; and \; x < y\}.$$

It is common to overload symbols such as $<$, which can be used to denote binary relations over Int (the set of positive and negative numbers) or $Real$. For the sake of uniformity, we permit the arity of a relation to be 1. Such relations are called *unary* relations, or *properties.* For example, *odd* can be viewed as a unary relation

$$odd = \{x \;\mid\; x \in Nat \; and \; x \; is \; odd\},$$

or, equivalently,

$$odd = \{x \;\mid\; x \in Nat \; and \; \exists y \in Nat \;:\; x = 2y + 1\}.$$

[5] Our inability to deal with "even a three-line program" perhaps best illustrates Dijkstra's advice on the need to be *humble programmers* [36].

Much like we defined binary relations, we can define *ternary* relations (or 3-ary relations), 4-ary relations, etc. An example of a 3-ary relation over Nat is *between*, defined as follows:

$$between = \{\langle x, y, z\rangle \mid x, y, z \in Nat \wedge (x \leq y) \wedge (y \leq z)\}.$$

Given a *binary* relation R over set S, define the *domain* of R to be

$$domain(R) = \{x \mid \exists y : \langle x, y\rangle \in R\},$$

and the *co-domain* of R to be

$$codomain(R) = \{y \mid \exists x : \langle x, y\rangle \in R\}.$$

Also, the *inverse* of R, written R^{-1} is

$$R^{-1} = \{\langle y, x\rangle \mid \langle x, y\rangle \in R\}.$$

As an example, the inverse of the 'less than' relation, '$<$,' is the greater than relation, namely '$>$.' Similarly, the inverse of '$>$' is '$<$.' Please note that the notion of *inverse* is defined only for *binary* relations - and not for *ternary* relations, for instance. Also, *inverse* is different from *complement*. The *complement* of $<$ is $\geq$ and the complement of '$>$' is '$\leq$,' where the complementations are being done with respect to the universe $Nat \times Nat$.

For a binary relation R, let $elements(R) = domain(R) \cup codomain(R)$. The restriction of R on a subset $X \subseteq elements(R)$ is written

$$R\,|_X = \{\langle x, y\rangle \mid \langle x, y\rangle \subseteq R \wedge x, y \in X\}.$$

Restriction can be used to specialize a relation to a "narrower" domain. For instance, consider the binary relation $<$ defined over $Real$. The restriction $<\,|_{Nat}$ restricts the relation to natural numbers.

Putting these ideas together, the symmetric difference of '$<$' and '$>$' is the '$\neq$' (not equal-to) relation. You will learn a great deal by proving this fact, so please try it!

2.11 Functions as Relations

Mathematicians seek conceptual economy. In the context of functions and relations, it is possible to express all functions as relations; hence, mathematicians often view functions as special cases of relations. Let us see how they do this.

For $k > 0$, a k-ary relation $R \subseteq T_D^1 \times T_D^2 \times \ldots \times T_D^k$ is said to be *single-valued* if for any $\langle x_1, \ldots, x_{k-1} \rangle \in T_D^1 \times T_D^2 \times \ldots \times T_D^{k-1}$, there is at most one x_k such that $\langle x_1, \ldots, x_{k-1}, x_k \rangle \in R$. Any single-valued relation R can be viewed as a $k-1$-ary function with domain $T_D^1 \times T_D^2 \times \ldots \times T_D^{k-1}$ and range T_D^k. We also call single-valued relations *functional relations*. As an example, the ternary relation

$$\{\langle x, y, z \rangle \mid x, y, z \in Nat \wedge (x + y = z)\}$$

is a functional relation. However, the ternary relation *between* defined earlier is *not* a functional relation.

How do partial and total functions "show up" in the world of relations? Consider a $k-1$-ary function f. If a x_k exists for any input $\langle x_1, \ldots, x_{k-1} \rangle \in T_D^1 \times T_D^2 \times \ldots \times T_D^{k-1}$, the function is total; otherwise, the function is partial.

To summarize, given a single-valued k-ary relation R, R can be viewed as a function f_R such that the "inputs" of this function are the first $k-1$ components of the relation and the output is the last component. Also, given a k-ary function f, the $k+1$-ary single-valued relation corresponding to it is denoted R_f.

2.11.1 More λ syntax

There are two different ways of expressing two-ary functions in the Lambda calculus. One is to assume that 2-ary functions take a *pair* of arguments and return a result. The other is to assume that 2-ary functions are 1-ary functions that take an argument and return a result, where the result is *another* 1-ary function.[6] To illustrate these ideas, let us define function RMS which stands for *root mean squared* in both these styles, calling them rms_1 and rms_2 respectively:

$rms_1 : \lambda\langle x, y \rangle . \sqrt{x^2 + y^2}$
$rms_2 : \lambda x . \lambda y . \sqrt{x^2 + y^2}$

[6] The latter style is known as the *Curried* form, in honor of Haskell B. Curry. It was also a notation proposed by Schöenfinkel; perhaps one could have named it the 'Schöenfinkeled' form, as well.

Now, $rms_1\langle 2, 4\rangle$ would yield $\sqrt{20}$. On the other hand, we apply rms_2 to its arguments in succession. First, $rms_2(2)$ yields $\lambda y.\sqrt{2^2 + y^2}$, i.e., $\lambda y.\sqrt{4 + y^2}$, and this function when applied to 4 yields $\sqrt{20}$. Usually, we use parentheses instead of angle brackets, as in programming languages; for example, we write $rms_1(2, 4)$ and $\lambda(x, y).\sqrt{x^2 + y^2}$.

The above notations can help us write characteristic predicates quite conveniently. The characteristic predicate

$$(\lambda(z, y).(odd(z) \wedge (4 \leq z \leq 7) \wedge \neg y))$$

denotes (or, 'defines') the relation $= \{\langle 5, false\rangle, \langle 7, false\rangle\}$. This is different from the characteristic predicate

$$(\lambda(x, z, y).(odd(z) \wedge (4 \leq z \leq 7) \wedge \neg y))$$

which, for x of type $Bool$, represents the relation

$$R^{'} \subseteq Bool \times Nat \times Bool$$

equal to

$$\{\langle false, 5, false\rangle, \langle true, 5, false\rangle, \langle false, 7, false\rangle\}, \langle true, 7, false\rangle\}.$$

Variable x is not used (it is a "don't care") in this formula.

Chapter Summary

This chapter provided a quick tour through sets, numbers, functions, relations, and the lambda notation. The following exercises are designed to give you sufficient practice with these notions.

Exercises

2.1. Given the characteristic predicate $p = \lambda x.\ (x > 0\ \wedge\ x < 10)$, describe the unary relation defined by p as a set of natural numbers.

2.2. Given the characteristic formula $f = (x > 0\ \wedge\ x < 10)$, describe the unary relation defined by f as a set of natural numbers.

2.3. Given the characteristic predicate

$$r = \lambda(x, y, z).\ (x \subseteq y\ \wedge\ y \subseteq z\ \wedge\ x \subseteq \{1, 2\}\ \wedge\ y \subseteq \{1, 2\}\ \wedge\ z \subseteq \{1, 2\})$$

write out the relation described by r as a set of triples.

2.4. Repeat Exercise 2.3 with the conjunct $x \subseteq y$ removed.

2.5. What is the set defined by $P = \{x \in Nat \mid 55 < 44\}$?

2.6. The powerset, P, introduced earlier can also be written as

$$P = \{x \mid x \subseteq \{1, 2, 3\}\}$$

What set is defined by replacing $\subseteq$ by $\subset$ above?

2.7.

1. What is the set described by the expression

$$\{1, 2, 3\} \cap \{1, 2\} \cup \{2, 4, 5\}.$$

   Here, $\cap$ has higher precedence than $\cup$.
2. What is the *symmetric difference* between $\{1, 2, 3, 9\}$ and $\{2, 4, 5, -1\}$?
3. How many elements are there in the following set: $\{\emptyset\} \cup \emptyset \cup \{\{2\}, \emptyset\}$? $\emptyset$ denotes the empty set. It is assumed that sets may contain other sets.

2.8. Formally define the set S of divisors of 64. Either show the set explicitly or define it using comprehension.

2.9. Formally define the set S of divisors of 67,108,864. Either show the set explicitly (!) or define it using comprehension.

2.10. What is the set defined by $\{x \mid x \geq 0 \wedge prime(x) \wedge x \leq 10\}$?

2.11. What is the set defined by

$$\{x \mid x \in 13 \wedge composite(x) \wedge x \geq 1\}?$$

A composite number is one that is not prime.

2.12. What is the set defined by

$$\{x \mid x \in 25 \wedge square(x)\}$$

$square(x)$ means x is the square of a natural number.

2.13. What is the set $S = \{x \mid x \subset Nat \wedge 23 = 24\}$?

2.14. Take $S = Nat$, which contains an infinite number of elements. How many elements are there in the powerset of S? Clearly it also contains an infinite number of elements; but is it the "same kind of infinity?" Think for five minutes and write down your thoughts in about four sentences (we shall revisit this issue in Chapter 3).

2.15. The set Odd of odd numbers is a proper subset of Nat. It is true that Odd "appears to be smaller" than Nat - yet, both sets contain an infinite number of elements. How can this be? Is the 'infinity' that measures the size of Odd a 'smaller infinity' than that which measures the size of Nat? Again, express your thoughts in about four sentences.

2.16. Let E be the set of Even natural numbers. Express the set $E \times 2^E$ using set comprehension.

2.17. An *undirected graph* G is a pair $\langle V, E \rangle$, where V is the set of vertices and E is the set of edges. For example, a triangular graph over $V = \{0, 1, 2\}$ is

$$\langle \{0, 1, 2\}, \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 0, 2\rangle\}\rangle.$$

We follow the convention of not listing symmetric variants of edges - such as $\langle 1, 0\rangle$ for $\langle 0, 1\rangle$.

Now, this question is about *cliques*. A triangle is a 3-clique. A *clique* is a graph where every pair of nodes has an edge between them. We showed you, above, how to present a 3-clique using set-theoretic notation.

Present the following n-cliques over the nodes $i \in n$ in the same set-theoretic notation. Also draw a picture of each resulting graph:

1. 1-clique, or a point.
2. 2-clique, or a straight-line.
3. 4-clique.
4. 5-clique.

2.18. Write a function signature for the *sin* and *tan* functions that accept inputs in degrees.

2.19. Decipher the signature given below by writing down four distinct members of the domain and the same number from the range of this function. Here, X stands for "don't care," which we add to $Bool$. For your examples, choose as wide a variety of domain and range elements as possible to reveal your detailed understanding of the signature:

$$(Int \cup \{-1\}) \times 2^{Int} \times Bool \rightarrow 2^{2^{Bool \cup \{X\}}} \times Int.$$

2.20. Write a function signature for the function $1/(1-x)$ for $x \in Nat$.

2.21. Express the successor function over Nat using the Lambda notation.

2.22. Express the function that sums 1 through N using the Lambda notation.

2.23. Simplify $(\lambda zy.(\lambda x.(z + ((\lambda v.(v + x))5)))) \; 2 \; 3 \; 4$

2.24. A half-wave rectifier receives a waveform at its input and produces output voltage as follows. When fed a positive voltage on the input, it does not conduct below 0.7 volts (effectively producing 0 volts). When fed a positive voltage above 0.7 volts, it conducts, but diminishes the output by 0.7 volts. When fed a negative voltage, it produces 0 volts, except when fed a voltage below -100 volts, when it blows up in a cloud of smoke (causing the output to be undefined). View the functionality of this rectifier as a function that maps input voltages to output voltages. Describe this function using the Lambda notation. You can assume that $ifthenelse$ and $numbers$ are primitives in Lambda calculus.

2.25. Provide one example of a bijection from Nat to Int.

2.26. Point out which of the following functions can exist and which cannot. Provide reasons for functions that cannot exist, and examples for functions that can exist.

1. A bijection from $\emptyset$ to $\emptyset$.
2. A bijection from $\{\varepsilon\}$ to $\{\emptyset\}$.
3. A partial 1-1 and onto function from Nat to Nat.
4. A partial 1-1 and onto function from Int to Nat.
5. A 1-1 into function from Nat to Nat.
6. A 1-1 into, but not onto, function from Nat to Nat.
7. A bijection from Int to $Real$.
8. A bijection from a set to its powerset. (Recall that we cannot have $\emptyset$ as either the domain or range of a function.)
9. A many-to-one function from the powerset of a set to the set.
10. An into map from a set to its powerset.

2.27. Describe a bijection from the set $\{\varepsilon\}$ to the set $\{\emptyset\}$. Here, ε is the empty string.

2.28. Think about the following question, writing your thoughts in a few sentences, in case you cannot definitely answer the question (these will be addressed in Chapter 3).

- Can there be a bijection between Int and 2^{Int}?
- How about a finite subset, F, of Int, and 2^F?
- How about an infinite subset, I, of Int, and 2^I?
- Which other kinds of functions than bijections may exist?

2.29. Do you see any problems calling a function g "computable" if g were to accept a subset of Nat, output 4 if given $\{1, 2\}$, and output 5 given any other subset of Nat? How about a variant of this problem with "any other subset" replaced by "any other *proper* subset?"

2.30. Which of the following functions are computable?:

1. A function that inverts every bit of an infinite string of bits.
2. A function that inverts every bit of a finite (but arbitrarily long) string of bits.
3. A function that outputs a 1 when given π, and 0 when given any other Real number. (Recall that π is not 22/7, 3.14, or even 3.1415926. In fact, π is not a Rational number.)

2.31. Does there exist a procedure that, given a C program P and its input x, answers whether P halts on x? Does there exist an algorithm for this purpose?

2.32. Can there be an algorithm that, given two C programs, checks that they have identical functionality (over *all* their inputs)?

2.33. Can there be an algorithm that, given two `Yacc` grammar files (capturing context-free grammar productions) checks whether the grammars encode the same language or not? (`Yacc` is a tool to generate parsers). Write your 'best guess' answer for now; this problem will be formally addressed in Chapter 17.

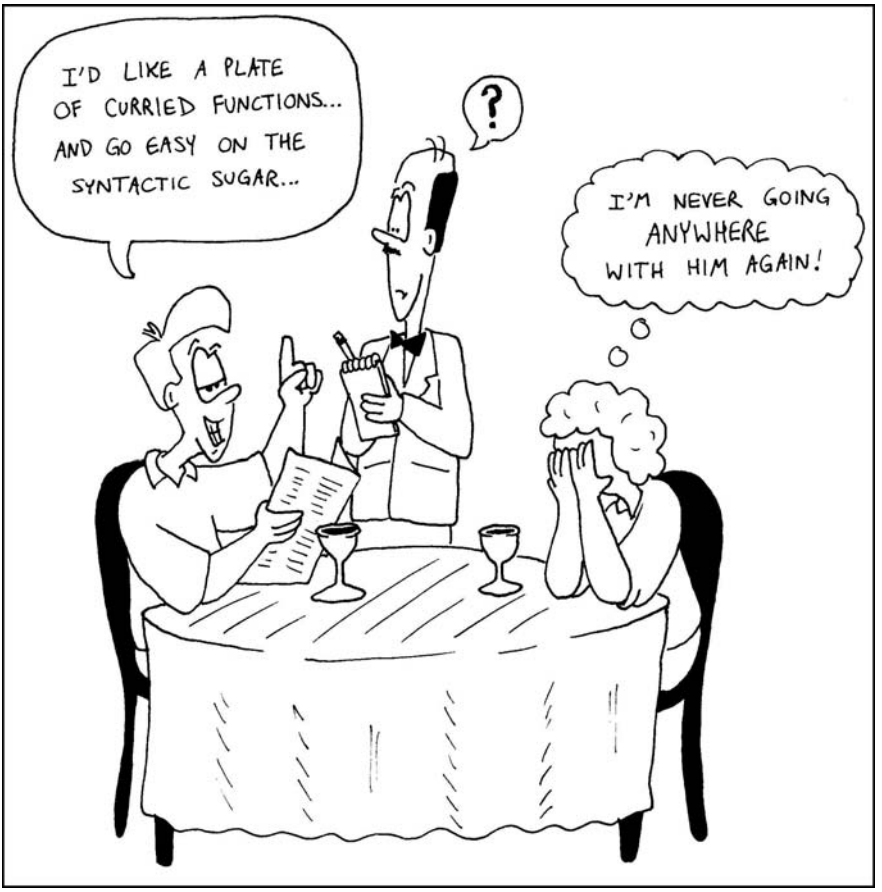
2.34. What is the symmetric difference between '$\leq$' and '$\geq$'? How about the symmetric difference between '$<$' and '$\leq$'?

2.35. Consider the binary relation *relprime* over $Nat \times Nat$ such that $relprime(x, y)$ exactly when x and y are relatively prime (the greatest common divisor of x and y is 1). Is *relprime* a functional relation? What is its *inverse*? What is its *complement*?

2.36. Consider the 3-ary relation "unequal3," which consists of triples $\langle a, b, c\rangle$ such that $a \neq b$, $b \neq c$, and $a \neq c$. Is this relation a functional relation? Provide reasons.

2.37. How many functions with signature $Bool^k \rightarrow Bool$ exist as a function of k? Here $Bool^k$ is $Bool \times Bool \times \ldots \times Bool$ (k times). Think carefully about *all possible distinct functions* that can have this signature. Each k-ary Boolean function can be presented using a $k + 1$-column truth table with each row corresponding to one input and its corresponding output.

2.38. Repeat Exercise 2.37 for partial functions with signature $Bool^k \rightarrow Bool$. View each partial function as a table with the output field being either a Boolean or the special symbol $\perp$, standing for *undefined.*



http://www.springer.com/978-0-387-24418-1

Computation Engineering
Applied Automata Theory and Logic
Gopalakrishnan, G.
2006, XXXVI, 472 p., Hardcover
ISBN: 978-0-387-24418-1