Chapter 3

Functions and Proofs

3.1 Functions

A *function* (or a *mapping*) is a mathematical object that sets up an input-output relationship. It takes an input (or an argument) and returns an output (or a result). If f is a function that, when it receives an input a, produces the result b, we write $f(a) = b$. We also say that f *maps* a to b, or that f *assigns* b to a. We also say that b is an *image* of a under f. A function is always defined on some set of inputs, also called its *domain*. The set from which the outputs of a function come is called its *range*. If a function f has domain D and range R, we write $f : D \to R$. Thus, for example, we can define a function square that takes as an input an integer number and produces as an output the square of that number. We can say that the domain of this function is the set of integer numbers, and that its range is also a set of integer numbers (or we can be more precise and say that the range is the set of all non-negative integer numbers). We can write this as square : $\mathbb{Z} \to \mathbb{Z}$.

We can observe many similarities between the concept of mathematical function and the concept of functions or methods in any of the programming languages. For example, C++ function or Java methods also take an input (a list of parameters, to which concrete values are assigned in each invocation of the function) and produce an output (which is possibly void). Functions of multiple arguments are simply just functions that take a tuple as an argument. For example, the function add that takes two integer numbers and adds them can be defined on a domain that is the set of all pairs of integer numbers, and its range is the set of integer numbers. There are, however, three key differences between mathematical functions and functions in most of the programming languages:

- The mathematical function has to be defined for every element of its domain.
- The mathematical function has to return a single element of its range for each input.
- The mathematical function has to return the same single output for the same input.

Because of the third feature of mathematical functions, there is no equivalent of the concept of modifying the external state or the environment (like, for example, modifying global variables) as it exists in most of the programming languages. Some of the most interesting functions in programming languages return void (or equivalent), and exist solely for the purpose of modifying the environment external to the function in some way (e.g. by printing on the screen). Mathematical function that always returns the equivalent of void would be hopelessly uninteresting (akin to a function that returns 0 for every possible input).

Let us now look at some examples (and non-examples) of functions:

- $f_1 : \mathbb{Z} \to \mathbb{Z}$, where $f_1(x) = x + 1$. $\mathbb{Z}$ is the set of all integer numbers.
- $f_2 : \mathbb{Z} \to \{0, 1\}$, where $f_2(x)$ is 0 if x is even, and 1 otherwise.
- $f_3 : \Sigma^* \to \Sigma^*$, where $f_3(w) = ww$, for each string w from Σ^*.
- $f_4 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, where $f(x, y) = x + y$.
- *Nonexample*: $f_5 : \mathbb{Z} \to \{0, 1\}$, $f_5(x)$ is 0 if x is divisible by 2 and $f_5(x)$ is 1 if x is divisible by 3.

The last example is worth special attention. Why is f_5 not a function? Firstly, it is not defined for every element of $\mathbb{Z}$. It is not defined, for example, for $x = 5$. But this could be solved if we consider a different domain, for example the set of all integer numbers that are divisible by 2 or divisible by 3. For example, if we we define a set $X = \{0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, \ldots\}$, we can then define $f_5 : X \longrightarrow \{0, 1\}$. Now f_5 is defined for each element of its domain. However, it is still not a function, as it returns both 0 and 1 for an input 6, whereas for an input 2 returns only 0. The function has to return a single element of its range for every input. One way we can "fix" f_5 to be a function is to define it so that it returns a set of numbers instead of just one number. Thus, it would return the set $\{0\}$ if the input is divisible by 2, the set $\{1\}$ if the input is divisible by 3 and the set $\{0, 1\}$ if the input is divisible by both 2 and 3. The range of the function would then be the set $\{\{0\}, \{1\}, \{0, 1\}\}$. In this case the function would, indeed, return a single element of its range for each input, and it would return the same single output for the same input.

When we define a function, we have a certain freedom in choosing its domain, because the domain of the function is an integral part of its definition. For example, it is not enough to just say that we will define an addition function as a function that returns the sum of two numbers that are its arguments. We also need to clarify whether we define this function for the domain of integer numbers, rational numbers, real numbers, functions themselves or perhaps some subset of these. The addition function on the domain of integer numbers is a completely different function from the addition function on the domain of real numbers, which itself is a different function from the addition function on the domain of the real numbers between 0 and 1, both included. However, most of the time when we define a function, we assume that its domain is a maximum set (in terms of the subset relation) for which the definition makes sense. For example, when looking at the addition function, most of the time we implicitly assume that we are considering this function on the set of all real numbers. Such a domain will be called a *natural domain* of a function. Some examples of natural domains of functions are:

- $f_1(x) = x + 1$, when seen as a function on integer numbers, has as a natural domain the set all integer numbers $\mathbb{Z}$.
- $f_2(w) = ww$, for some word w over the alphabet Σ, has as a natural domain the set of all strings over Σ (Σ^*).
- $f_3(x) = \frac{x}{x+1}$. when seen as a function on real numbers, has as a natural domain the set of all real numbers other than -1 ($\mathbb{R} \setminus \{-1\}$).

Definition 3.1: Equality of Functions

Two functions, $f : D_1 \longrightarrow R_1$ and $g : D_2 \longrightarrow R_2$, are equal if they have the same domain and if, for each element of that domain, they return the same value.
More formally, $f = g$ if $D_1 = D_2$ and $f(x) = g(x)$ for every $x \in D_1$.

For example, according to this definition, functions $f(x) = x$ and $g(x) = 2x - x$ on real numbers are equal when we take their natural domains as domains. On the other hand, functions $f(x) = x$ and $g(x) = \frac{2x^2}{x}$ on real numbers, when considered on their natural domains, are not equal. This is because they have different natural domains. f has as a natural domain the set $\mathbb{R}$ and g has a natural domain the set $\mathbb{R} \setminus \{0\}$. They, however, return the same value on the inputs from the set $\mathbb{R} \setminus 0$, so if we take as their domains this set, they would be equal.

Most of the times, we describe functions by describing the procedure for computing the output from an input (e.g. $f(x, y) = x + y$). But sometimes this can also be done by using a table, especially if the function domain is small (in terms of the number of elements in it. Consider $f : \{0, 1, 2, 3\} \longrightarrow \{1, 2, 3, 4\}$ where $f(x) = x + 1$. This function can be described by the table

x	$f(x)$
0	1
1	2
2	3
3	4

Consider also $f : \{0, 1, 2\} \times \{0, 1, 2\} \longrightarrow \mathbb{Z}$ where $f(x, y) = x + y$. This function can be described by the table

f	0	1	2
0	0	1	2
1	1	2	3
2	2	3	4

3.2 Recursive Definitions

So far in our discussion, all the definitions that we have introduced have been 'direct' and relatively simple. But sometimes, it is very convenient to introduce an object in terms of how it can be composed from the small instances of that same objects, down to some base cases. In this kind of a definition, we first specify one or more *basic objects*, and then we give rules for constructing more complex objects from the ones that we already know. These are so called *recursive definitions*, and we are going to use them in a few places in this module (see, for example, the definition of a regular expression in Section **??**).

Let us now see some examples of the recursive definitions. Positive even integer numbers can be defined as numbers that are divisible by 2. This is perfectly good definition and, in fact, the most useful one in many instances. But it is also possible to define positive even integer numbers in a recursive way:

1. 2 is an even number.
2. If x is an even number, then $x + 2$ is also an even number.

This is a more complicated definition, but in some cases more useful. In other cases, recursive definitions are not only more useful, but also much simpler and more intuitive. Take, for example, a definition of an arithmetic expressions. Let us suppose that we know what numbers are and what arithmetic operators $*$, $/$, $-$ and $+$ are. How would we then define an arithmetic expression? Some valid arithmetic expressions are $2 + 3$, $2 - 3/4$, $(2 + 3) * 4$, $2 - 3 * 4$, $2 - (-3)$, while $2 + -3$, $2//$, $2) + (3$, $2-$ are not valid. Defining directly a valid arithmetic expression is far from trivial. But consider the following recursive definition:

1. Any number x is an arithmetic expression.
2. If x is an arithmetic expression, then (x) and $-(x)$ are also arithmetic expressions.
3. If x and y are arithmetic expressions, then so are
   (i) $x + y$ (if the first symbol in y is not $-$)
   (ii) $x - y$ (if the first symbol in y is not $-$)
   (iii) $x * y$
   (iv) x/y

This definition is more intuitive, but also more instructive, as it gives us a method of constructing (and deconstructing) arithmetic expressions. Thus, for example, it gives us a precise way of constructing the arithmetic expression $(2 + 3)/4$, by first applying the rule 3i to get the arithmetic expression $2 + 3$ from the numbers 2 and 3 (which are themselves arithmetic expressions, based on the rule 1), then applying the rule 2 to get the arithmetic expression $(2 + 3)$ and finally applying the rule 3iv to this arithmetic expression and number 4. This makes it easier to do formal reasoning about arithmetic expression than if we used a non-recursive definition.

As another example, let us take palindromes over the alphabet $\{0, 1\}$. We can define the set of palindromes P over $\{0, 1\}$ by

$$P = \{w|w \text{ is a string from } \Sigma^*, w = w^R\},$$

where w^R is the string w reversed. This is pretty simple and intuitive, but it is not very easy to prove that something is a palindrome using this definition. On the other hand, let us consider the following recursive definition:

1. ϵ, 0 and 1 are all palindromes.
2. If x is a palindrome, so are $0x0$ and $1x1$.

Again, this gives us not only an equally intuitive definition, but also a method for constructing (or deconstructing) palindromes. It is much easier to prove that some string w is a palindrome using this definition.

3.3 Definitions, Statements, Proofs and Theorems

A *definition* is used to describe *precisely* the objects and notions that we use. The concept of a definition might seem simple and obvious enough at first, but actually it can be quite subtle. Firstly, the definition has to be absolutely precise and must not leave any room for different interpretations. Secondly, the definition of some new concept can only use the terms that have been introduced before. Taking, for example, the Definition 2.8 of a language closure, the new term “language closure” there relies on previously introduced concepts such as “language”, “string” and “string concatenation”. If any of them was not precisely defined before, the definition would not have been valid. The definition also contains an assumption that L is a language. It is perfectly fine to assume something in a definition, keeping in mind that some object conforms to the definition only if all the assumptions from the definition for that object are satisfied.

Statements typically express that some object has some property. They also have to be precise and have to be classifiable as true or false. Some examples are

- *It rained in Dundee yesterday* is a statement.
- *At some point in future, the Earth will be hit by a giant asteroid* is a statement.
- *How are you?* is not a statement.

The first sentence is a statement, provided that we precisely define what do we mean by ’rained in Dundee’. Do we mean in at least one spot within the city boundary of Dundee or do we mean in every spot? Once we clarify this, the statement becomes precise enough so that it can be classified as true or false. In the second example, we need to define what do we mean by ’a giant asteroid’, but provided we do that, the sentence becomes a statement. Depending on how we define a giant asteroid, We cannot, at this point in time, determine whether the statement is true or false, nor will we (most likely) ever be able to establish that (except in the Armageddon-like scenario, where such event becomes inevitable). But the fact that we cannot determine whether the statement is true or false does not mean that it is not a statement, since it definitely either true or false.

A *proof* is a convincing logical argument that a statement is true. A big part of this module is proving various statements about the computing models (finite state automata and Turing machines) that we will consider. A proof is a sequence of statements, each of which is either a definition or logically follows from the previous statements. "Logically follows" cannot be open to interpretation - it has to be absolutely precise and beyond any doubt. A *theorem* is an important (in some sense) mathematical statement that is proved to be true. Less important statements proved to be true are usually called *lemmas*.

Remark 3.1: Notation for the End of a Proof

In this text, we will denote the end of a proof with a square symbol at the end of the line, such as here. □

Let us now show how proofs work on one example.

Theorem 3.1: DeMorgan’s Law

For any two sets A and B, $\overline{A \cup B} = \overline{A} \cap \overline{B}$.

Proof. To prove this theorem, we first need to understand what it states. We need to know what $\cup$, $\cap$ and overbar operations mean. These were introduced in Definitions 1.3, 1.6 and 1.11. The theorem states the equality between the two sets, $\overline{A \cup B}$ and $\overline{A} \cap \overline{B}$. How do we prove equality between the two sets? Recall the definition **??**. To prove the equality of two sets, we need to prove that the first set is a subset of the second set and that the second set is a subset of the first set. So, the proof here is split into two parts.

So, let us take any two sets A and B. Let us first prove that $\overline{A \cup B} \subset \overline{A} \cap \overline{B}$. We need to prove that every element of $\overline{A \cup B}$ is also an element of $\overline{A} \cap \overline{B}$. Let x be an arbitrary element of $\overline{A \cup B}$. This means (by definition of a set complement) that x is **not** in $A \cup B$. This further means (by definition of a set union) that x is **not** in A and x is **not** in B. x is not in A means that (by definition of a set complement) x is in $\overline{A}$. Similarly, x is not in B means that x is in $\overline{B}$. Since x is both in $\overline{A}$ and $\overline{B}$, it means (by definition of set intersection) that x is in $\overline{A} \cap \overline{B}$. Since we took x to be an arbitrary element of $\overline{A \cup B}$, it means that this holds for *every* element of $\overline{A \cup B}$. So, every element of $\overline{A \cup B}$ is also an element of $\overline{A} \cap \overline{B}$. This means (by the definition of a subset of a set) that $\overline{A \cup B} \subset \overline{A} \cap \overline{B}$.

Let us now prove that $\overline{A} \cap \overline{B} \subset \overline{A \cup B}$. We need to prove that every element of $\overline{A} \cap \overline{B}$ is also an element of $\overline{A \cup B}$. Let x be an arbitrary element of $\overline{A} \cap \overline{B}$. This means (by definition of a set intersection) that x is in $\overline{A}$ and x is in $\overline{B}$. This further means (by definition of a set complement) that x is **not** in A and x is **not** in B. If x is not in A and x is not in B, it means (by the definition of set union) that x is not in $A \cup B$. If x is not in $A \cup B$, it means that x is in $\overline{A \cup B}$. Since we took x to be an arbitrary element of $\overline{A} \cap \overline{B}$, it means that this holds for *every* element of $\overline{A} \cap \overline{B}$. Hence, every element of $\overline{A} \cap \overline{B}$ is also an element of $\overline{A \cup B}$, which means that $\overline{A} \cap \overline{B} \subset \overline{A \cup B}$. Taking into account that in the first part we proved that $\overline{A \cup B} \subset \overline{A} \cap \overline{B}$, this means (by the definition of set equality) that $\overline{A \cup B} = \overline{A} \cap \overline{B}$. □

3.4 Types of Proofs

In this section, we will briefly describe the main types of proofs. Whilst we are not going to fully formally prove many of the statements in this text, it is still useful to be aware of the general proof techniques that are used in mathematics. Of particular importance in Theory of Computation is a *proof by induction* (Section 3.4.3), as most of the statements are proved using this technique.

3.4.1 Proof by Construction

Many statements state that a particular object exists. At many places in this text, you will find the statements of type "There exists a finite automata such that..." or "There exists a Turing Machine such that...", followed by one or more properties. One way to prove such a statement is to explicitly show how to construct such an object, and then prove that this object indeed has the required properties. This is a *proof by construction* and it is a particularly elegant and rewarding kind of a proof, because it not only proves that an object with required properties exist, but also gives us a recipe how to construct such an object. Whenever possible, we should prefer these kind of proofs.

3.4.2 Proof by Contradiction

In *proof by contradiction*, we assume that the theorem is false and then show that this assumption leads to an obviously false consequence. This is most commonly used to prove the statements of the type "If A is true, then B is also true". We then assume that A is true and that B is false, and usually arrive at the conclusion that A cannot be true, which is then a contradiction (because we assumed that A is true). This argument relies on the fact that the statement can be either true or false. We eliminate the case that the statement is false (because it led us to contradiction), so the only remaining option is that the statement is true. This is perfectly valid, but somewhat less rewarding proof method. It can also be used in some cases to prove an existence of an object with some properties, by showing that the assumption that an object with those properties does not exist leads us to some contradiction, but it would not give us any method of actually constructing such an object. It would just prove its existence (which is enough for the purpose of proving the statement).

This kind of a proof is what we very often employ in our everyday reasoning. For example, imagine the following situation. Jack sees Jill coming in from the outdoors. Jill is completely dry. Jack wants to go out, but before that he wants to establish that it is not raining outside. That is, he wants to prove the statement "It is not raining outside". Jack first assumes that this is false and that, therefore, it is raining outside. But if it was raining outside, Jill would not have been completely dry. This is a contradiction to the fact that Jill is completely dry. Therefore, assuming that it is raining outside led Jack to a contradiction. Since it can either rain or not rain outside, and since Jack has established that the statement "It is raining outside" is false, it can only mean that it is not raining outside - there is no third option. Note that Jack did not directly establish that it is not raining outside (for example, by looking through the window), but merely that the assumption that it is raining outside is false.

3.4.3 Proof by Induction

Proof by Induction is a slightly more complicated, but very common proof technique, especially in Computer Science. It is very often tightly tied to recursive definitions. The basic form of induction is used to prove properties that are parameterised over natural numbers (possibly including 0). We denote by $P(n)$ a property parameterised by a natural number. Some examples of $P(n)$ are

- "The sum of first n natural numbers is $\frac{n(n+1)}{2}$.

- "If a sequence of integer numbers of length n, then Quick Sort sorting algorithm can sort it in polynomial time with respect to n."

A more complicated example relies on a recursive definition. Let us recursively define the language L over the alphabet $\Sigma = \{0, 1\}$:

1. Strings ϵ belongs to L.
2. If a string w belongs to L, so does the string $w001$.

The following is then an example of a statement $P(n)$: "Any string of length $3n$ from the language L has $2n$ 0's in it.".

To prove the property $P(n)$ by induction we proceed in two parts:

1. Prove that $P(n)$ is true for $n = 1$ (or $n = 0$). In other words, prove that $P(1)$ (or $P(0)$) holds. (*base case*)
2. Prove that *if* $P(k)$ holds, *then* $P(k + 1)$ holds too. (*induction step*)

Note that in the induction step, we make an assumption that $P(k)$ holds, and then using this assumption prove that $P(k + 1)$ holds too. Why does all this prove that the property holds for every n?. For start, because we have proved the base case, we know that $P(1)$ holds. Since we know that $P(1)$ holds, by induction step we also know that $P(2)$ holds. That is, we know that *if* $P(1)$ holds, *then* $P(2)$ also holds, and we know that $P(1)$ holds. Therefore, it has to be the case that $P(2)$ holds. Using the same reasoning, since we know that $P(2)$ holds, by induction step we know that $P(3)$ holds too. We proceed by this argument and conclude that $P(n)$ indeed holds for every n.

Let us now seen an example of a proof by induction. Let us prove the following theorem.

Theorem 3.2: Sum Of The First n Natural Numbers

The sum of the first n natural numbers, denoted by $S(n)$, is $\frac{n(n+1)}{2}$.

Proof. We prove this by induction. Our statement $P(n)$, parameterised by n, in this case is "The sum of the first n natural numbers is $\frac{n(n+1)}{2}$" or, more concisely, $S(n) = \frac{n(n+1)}{2}$.

Base case: $P(1)$ becomes "$S(1) = \frac{1\cdot 2}{2}$. $S(1)$ is obviously 1, as it is the sum of th first 1 natural numbers. $\frac{1\cdot 2}{2}$ is also obviously 1. Therefore, $P(1)$ is obviously true.

Induction step: If $P(k)$ holds, then $P(k + 1)$ holds translates into "if we know that $S(k) = \frac{k(k+1)}{2}$, then it is also true that $S(k + 1) = \frac{(k+1)(k+2)}{2}$". That is, we *assume* that $S(k) = \frac{k(k+1)}{2}$. Let us look at the sum of the first $k+1$ natural numbers. This $1+2+\cdots+k+(k+1)$ which can also be written as $(1+2+\cdots+k)+(k+1)$. The term in the first bracket is clearly the sum of the first k natural numbers, $S(k)$. But we have assumed that $S(k) = \frac{k(k+1)}{2}$. Therefore, we have

$$S(k + 1) = S(k) + (k + 1) = \frac{k(k + 1)}{2} + (k + 1) = \frac{k(k + 1) + 2(k + 1)}{2} = \frac{(k + 1)(k + 2)}{2}.$$

This is exactly what we needed to prove. Therefore, we have proved that, if $S(k) = \frac{k(k+1)}{2}$, then it has to be $S(k + 1) = \frac{(k+1)(k+2)}{2}$. Therefore, if $P(k)$ holds, then also $P(k + 1)$ holds. This also completes the inductive proof, therefore proving that $S(n) = \frac{n(n+1)}{2}$, for every $n \geq 1$. □

There are many variants of the induction proofs. We sometimes may have more than one base case. As an example, the induction proof has the following form:

- *Base case:* Prove that $P(0)$ and $P(1)$ hold.
- *Induction step:* Prove that, if $P(k)$ holds, then $P(k + 2)$ also holds.

Also, induction step does not always need to assume that $P(k)$ is true in order to prove that $P(k + 1)$ is true. Quite a common form of the induction is also:

- *Base case:* Prove that $P(1)$ holds.
- *Induction step:* Prove that if $P(i)$ holds for every $i < k$, then $P(k)$ also holds.

In Theory of Computation, many of the proofs are conducted using induction on the length of a string. Most commonly, we have to prove some property $P(w)$ of strings that belong to some language L. The way we usually proceed is to start from the shortest strings that belong to L (usually ϵ or single-symbol strings) as base cases, and then use induction on the length of a string that belongs to L. The induction proof then has a form:

- *Base case:* Prove that the property $P(w)$ holds for all strings w of minimal length (usually 0 or 1) that belong to the language L (usually 0 or 1).
- *Induction step:* Prove that if the property $P(w)$ holds for all strings w of length k from the language L, then it also holds for all strings v of length $k + 1$ from the language L.

Another very common kind of a statement that we prove by induction in Theory of Computation is some property $P(w)$ of all strings w over some alphabet Σ. In this case, the induction proof usually has the form:

- *Base case:* Prove that the property $P(w)$ holds for $w = \epsilon$.
- *Induction step:* Prove that if the property $P(w)$ holds for all strings w of length k, then it also holds for all strings w of length $k + 1$.