Organisation and Learning
Outcomes

Structure, Assessment, Coursework

All of this information is the same as for Part 1 and you should refer to those notes for the details.

Assumptions

This course requires a certain level of mathematical skill; indeed one of the goals is to show that there is a use for much of the discrete mathematics that you are taught elsewhere. Chapter 0 attempts to gather the assumed material together but you should also cross-reference material on discrete mathematics or use the web to look up unfamiliar concepts.

These Notes

This part of the course is divided up into four main chapters. Before they start there is an introduction to the Part which gives a high-level overview of the contents - I strongly suggest reading this.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read the introduction. I use these boxes to highlight things that are important that you may miss when scanning through the notes quickly. Typically the box will repeat something that has already been said. In this case, read the introduction!</td>
</tr>
</tbody>
</table>
Chapter 0 recaps some mathematical material from COMP11120. Then the main four chapters cover the four main concepts of models of computation, correctness, complexity and computability. There are examples and exercises throughout the notes. The assessed exercises for the examples classes are outlined in question sheets at the end of the notes. Some examples are marked with a (*) indicating that they go significantly beyond the examined material.

<table>
<thead>
<tr>
<th>Confused?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Great! Learning theory tells us that we do the best learning when we are confused by something and resolve that confusion. However, this only works if we manage to resolve the confusion. I try to use these boxes in the notes where I think something might be confusing. If there is somewhere where you get very confused and there is no box then let me know and I’ll add one.</td>
</tr>
</tbody>
</table>

Reading

In the following I suggest some textbooks that you might find helpful. However these notes should contain everything you need for this course and there is no expectation for you to read any of these books.

Although not examinable, some of you may be interested in the history of computation. There are many books and resources available on this topic. For example, Computing Before Computers now available freely online at http://ed-thelen.org/comp-hist/CBC.html

The following two books recommended from the first Part of the course are also relevant here. Hopcroft et al. cover computability (undecidability) and complexity (tractability) as well as Turing Machines (note that the 2nd Edition goes further into complexity). Parts 2 and 3 of Sipser cover computability and complexity respectively. Both books treat these topics quite theoretically, using Turing machines as models of computation and going beyond the contents of this course.


The following book covers complexity in a slightly more pragmatic way (Chapter 11) and was previously the recommended text for this topic.


For correctness see Chapter 4 of Huth and Ryan or Chapter 15 of Ben-Ari. Both books present the same axiomatic system we explore in these notes, and also cover a lot of things you might want to know about computational logic in general.


The above books also contain a lot of information that you can use to revise fundamental concepts that you should already know.

Those interested more generally in the semantics of programming languages could refer to the following. This material is well outside the scope of this course and these texts are only suggested for interest not for the course.


The last of which is out of print and available online at

Learning Outcomes

The course-level learning outcomes are as follows. Note that each chapter also includes chapter-level learning outcomes which give a more fine-grained overview of the intended learning outcomes of that chapter. At the end of this course you will be able to:

1. Explain how we model computation with the \texttt{while} language and the role of coding in this modelling
2. Write imperative programs in the \texttt{while} language and use the formal semantics of \texttt{while} to show how they execute
3. Prove partial and total correctness of \texttt{while} programs using an axiomatic system
4. Explain the notion of asymptotic complexity and the related notations
5. Use Big 'O' notation to reason about \texttt{while} programs
6. Explain notions of computability and uncomputability and how they relate to the Church-Turing thesis
7. Prove the existence of uncomputable functions
8. Describe why the Halting problem is uncomputable

Aims

In addition to the above learning outcomes I have the following broader aims for this material:

1. Communicate to you that theoretical computer science is not that difficult and has real applications
2. Introduce you to some concepts that I feel are essential for any computer scientist e.g. complexity analysis
3. Have fun

Acknowledgements

Thanks to Dave Lester for developing the previous version of these notes and to Joseph Razavi for providing feedback on parts of these notes.
Introduction to Part 2

Here I have chosen to introduce all of the concepts you will meet in this course at a high-level in a single (shortish) discussion. Hopefully this will give useful context and provide adequate motivation for the material presented. I suggest you reread this section a few times throughout the course and hopefully by the end of the course everything will be clear.

What is Computation?

This part of the course is about computation, but what is it? We are all familiar with computers and clearly computers are things that compute. But what does that mean? In Table 3 (see overleaf, sorry I couldn’t get it on this page) I give two alternative introductions to this question, one from the practical perspective and one from the theoretical/abstract perspective. I have purposefully written this side-by-side to stress that one perspective is not necessarily more important than the other. Both perspectives conclude that we need a model of computation with which to reason about what computers do, or what computation is.

If a computation is a series of instruction executions or symbol manipulations then a static description of a computation is a list of the instructions or operations that we need to perform to achieve that computation. We should be familiar with such descriptions as programs. Independent of the model of computation we chose we can abstract computations as programs.

Now that we have decided that we are talking about programs as static descriptions of computations we can ask some interesting questions, such as:

- Is the program correct?
- Which programs are better at solving a given problem?
- Are there things we cannot compute i.e. problems for which we cannot write a program?
The Practical Perspective
A computer has a processor that executes instructions and stores data in registers. But the processor in my mobile phone is different from the one in my laptop and also different from what we might find in a washing machine or supercomputer. It is clear that all of these processors are doing broadly the same thing; they are computing. But we want to be able to talk about this computation abstractly, without worrying about the exact instructions available or how registers are accessed. To achieve this we want a model that has an abstract notion of stored data and instruction execution and we want this model to be general enough that we can convince ourselves that we could use it to implement any of the concrete computational devices available.

The Abstract Perspective
At a high level of abstraction, computation is the process of performing some operations to manipulate some symbols to produce a result. When we add two numbers together in our head the operation is mental and the symbols abstract, when we do it on paper the operations involve a pen and the symbols are lines forming numbers, and when a computer performs the addition it applies logic gates to binary digits. In each case the result is clear. But the computation being performed was not dependent on the operations or symbols involved. There is an abstract notion of what it means to compute which is not related to the mechanisms by which the computation is carried out. To talk about this abstract notion of computation we want a model that captures this idea of performing operations on symbols.

Table 3: Two perspectives on what computation is.

These are the topics covered by the later parts of the course. But before we can discuss these topics we want a concrete model of computation in which to write programs.

A Model of Computation
There exist various models of computation and you have probably heard of some of them. The most famous is perhaps the Turing Machine, introduced by Allan Turing in 1948. This is an abstract machine that operates on an infinite tape reading and writing symbols based on a finite table of instructions (we will revisit this again later in Section 4.6). We will also
hear about some other more traditional models of computation, but we will not be using them to discuss computation.

In this course we will use a very simple programming language, called while. This kind of language should be very familiar to you by now, because all of its features are available in other programming languages such as java and python. The language is not that important, what is important is that it is simple and able to capture all computable functions, which it does as we find out later. Indeed, in this course we are not interested in how to write specific programs in this language (although you will be doing that) but about methods for analysing programs in this language (for their correctness and complexity) and properties of the language itself (for which we might need to write some programs to establish such properties).

The key feature informing the design of this new language (while) is its simplicity when compared to real programming languages. For example there is no inheritance, no data structures (other than the integers ($\mathbb{Z}$), which are not just the 32-bit variety), no functions or methods (indeed no way to structure programs at all), and not even any way to document the code with comments! This considered, this language is completely unusable for real world programming and software engineering.

However, it has some positive points:

- The while language is very small: there are five sorts of statements, six sorts of boolean expression, and five sorts of arithmetic expression. Despite this, it is possible to write any program in this language we could instead write in java or python.

- The above minimality means that we can define the language precisely in a reasonable amount of space and provide the rules needed to prove the correctness of programs in the language. For more complicated languages such definitions quickly become very large and difficult to explain.

- The while language looks like a proper programming language, and indeed the techniques that will be described for reasoning about the language are applicable to the more complicated programming languages which you will use throughout your career.

The key point to take away from the list above is that as we make the language more usable, it becomes more complicated, and the simple proofs we will be doing about the language itself will become much more complicated.
One of the reasons we have chosen to use the \texttt{while} language instead of Turing machines is that the language is a reasonable abstraction of the programs you will write, meaning that it makes sense to discuss concepts such as correctness and complexity. Whilst Turing machines are the common model of computation used when discussing computability, they are not very useful for presenting these other topics. Importantly, we don’t lose anything by choosing the \texttt{while} language. Due to the Church-Turing thesis (covered in Chapter 4) these models are \textit{equivalent}. This is important so I will write it in a box.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Important} \\
\texttt{The while} language has the same computational power as Turing machines; it is \textit{Turing-complete}. \\
\hline
\end{tabular}
\end{center}

\textbf{What Are We Computing?}

So we have a language in which to write programs and this language is our model of computation. But a program is just a series of instructions, it tells us nothing about what is being computed. But we do know that a program maps some inputs to outputs. In this sense it computes a \textit{function}\footnote{Some real world programs compute \textit{relations} instead of functions i.e. given a single input multiple outputs are possible i.e. the program is non-deterministic. We do not consider such programs in this course.}. The \textit{domain} of the function is the type of inputs it expects and the co-domain (range) is the type of outputs it produces.

In this course we will mostly restrict our attention to functions on natural numbers ($\mathbb{N}$) i.e. functions in $\mathbb{N} \to \mathbb{N}$. We will do this for two reasons:

1. It is easier. Proofs about computable functions are a lot easier to perform in this domain. Programs that take one output and return one output are easier to write - although in general our programs will compute functions in $\mathbb{N}^a \to \mathbb{N}^m$ i.e. tuples of numbers to tuples of numbers, as that is easier. When trying to grasp complex concepts it is best to keep things as simple as allows for the concept to still make sense.

2. We can \textit{code} any more interesting domain into $\mathbb{N}$. For example, we will see (Section 1.4.1) how to code pairs of numbers as numbers.
This argument can be extended to any finite structure. So by only considering functions on natural numbers we do not lose anything.

In fact, an important result in computability theory is the parameterisation theorem (Section 4.5.3) which tells us that we can take any computable function with \( n \) parameters and turn it into a function of \( n - 1 \) parameters (for \( n > 1 \)).

You might notice that my argument that we can encode anything we are interested in using \( \mathbb{N} \) does not allow us to represent functions on \( \mathbb{R} \) (the real numbers) as there is no way to represent all numbers in \( \mathbb{R} \) using \( \mathbb{N} \) (because \( \mathbb{R} \) is uncountable). But I am happy with this - show me a computer that can represent \( \mathbb{R} \) and I will update my definitions. Practically we can only represent finite approximations of real numbers and we encode any finite structure in \( \mathbb{N} \).

We might also talk about programs being solutions to problems where a problem might have different solutions. For example, the well-known sorting problem has various well-known solutions. So what are these problems and how do they relate to this idea that we compute functions? Talking about a problem is just an informal (lazy) way of talking about a function. The sorting problem is the function that takes a list and produces a list that is sorted. But we will still use the term problem as it is nice and friendly.

<table>
<thead>
<tr>
<th>Confused?</th>
</tr>
</thead>
<tbody>
<tr>
<td>At this point you might be a little confused. We have all these new concepts and how are they related? Is a computation a program, how is a model of computation related to these things? In attempt to clarify these points let us consider an analogy. We begin by noting that compute is an action-word (verb) so it describes something that is active. Let us consider an analogous action of make a cup of tea (not a single word, but you get the idea). So if compute is the verb then the noun computation describes the action (or set of actions) that was performed i.e. it describes the active thing that happened. So in our analogy a computation is a description of a single tea making process. A computation computes a function i.e. it transforms some start stuff to some end stuff. In the analogy we go from not having a cup of tea to having a cup of tea. The function is abstract and does not tell us anything about how we are supposed to get from the start stuff to the end stuff (i.e. how to make a cup of tea).</td>
</tr>
</tbody>
</table>
A program is a set of instructions that we can follow to perform a computation (we could replace program by algorithm). So if we follow our *Cooking for Undergraduates 101* recipe for making a cup of tea then we have used a program.

The difference between a program and a computation is that the program is the static thing and the computation is the active thing. Finally, a model of computation is the language in which we describe programs, we need this to make our programs unambiguous and to relate them to functions. We don’t have a model of computation in our analogy because we are not that overly concerned about making sure the semantics of tea-making recipes are well-defined.

Is My Program Correct?

---

I am using java programs in this motivational introduction as these are programs you should be able to reason about already. Note that the language used in this course, and importantly in exercises, is the *while* language introduced in Chapter 1.

---

It shouldn’t be surprising that we want the programs we write to be correct. But what does that mean? Is the following java program correct?

```java
public int f(int[] array){
    int len = array.length;
    int sum = 1;
    for(int i=0;i<len;i++){
        sum = sum*array[i];
    }
    return sum;
}
```

You might immediately spot that it is not syntactically correct, on line 2 we are missing a semi-colon. Those of you used to such errors might also spot the out-by-one error in the loop where we use \( \leq \) instead of \(<\). If we fix
these two errors is the program correct? If your answer is yes then you are wrong. However, if your answer is no then you are also wrong because the question doesn’t make sense! Sorry about that - in general I will try and avoid asking questions that don’t make sense, but sometimes they can be useful to encourage us to reflect on what such questions really mean.

For something to be correct we need to know what it is supposed to do and then we can check whether it does this. It might appear that the code is supposed to compute the sum of the values in array, as we use a variable called sum. Does the program do this? No, it computes the product of the values in array. Whether this is correct or not depends on the specification. Although using a variable called sum when we’re computing the product is probably a bad idea.

You should have met the idea of testing. Let’s assume the program is supposed to compute the product, what is the specification? We can define this using a precondition and postcondition. A precondition states the inputs that we think that the program is correct for. Here we can use an empty precondition as the program should work for all arrays. The postcondition states what the output should look like, here we can informally write the product of the values in the input array but later we will want to do this more formally. Notice that this pre/postcondition format also specifies the domain and co-domain of the function being implemented by the program.

Now given the above program we can test it by picking input values that make the precondition true and checking that the result is true for the postcondition. Try typing it into your computer and running it with array = [3,2,1] does it give the right answer? What other inputs should we try? I could have written the program

```java
public int f(int[] array){
    int len = array.length;
    int sum = 1;
    for(int i=0;i<len-1;i++){
        sum = sum*array[i];
    }
    return sum;
}
```

which would still have computed the correct product of array = [3,2,1], so we should also try array=[1,2,3]. Are we convinced that the program will compute the correct results on all inputs yet? In the extreme case we
cannot be convinced without trying them all\textsuperscript{2}. What we really want to be able to do is prove the program is correct.

This is what we do in Chapter 2. We will introduce a small proof system that allows us to build proofs that \texttt{while} programs are correct i.e. that given any state satisfying the program’s precondition, the program will transform that state into one satisfying the program’s postcondition.

There is one extra complication to this puzzle. What about the following program for computing the product of an array?

```
public int f(int[] array){
    int product = 1;
    bool isProduct = false;
    do{
        product = random.nextInt();
        isProduct = true;
        int check = product;
        for(int i = 0; i < array.length; i++){
            // first check that array[i] divides product evenly
            if(product % array[i] != 0){
                isProduct = false;
                break;
            }
            check = check / array[i];
        }
    } while(!isProduct);
    return product;
}
```

With some effort we can convince ourselves that if this method returns then it returns the product of the values in \texttt{array} but will it eventually return? It is not guaranteed that after some finite amount of time we will correctly guess the right answer, so it is possible that this method will continue trying to guess the product of the values in \texttt{array} forever!

\textsuperscript{2}The extreme case means large and complex programs. Note that if we have access to the source code then we can apply methods to ensure that we have covered enough of the program’s control and data structures. But these methods are similar in complexity to the ideas we talk about here, and can often only give an approximation of coverage.
In Chapter 2 we separate the question of whether a program is correct with respect to its specification *whenever it terminates* from the question of whether the program terminates. The former notion of correctness is called *partial correctness*, as it only applies to terminating executions of the program, and the latter is called *total correctness*.

Some of you at this point will have heard about the *Halting Problem* and be wondering how it relates to this idea of proving whether programs terminate. We meet the Halting Problem later in the course as it is an important concept in computability theory. However, at this point it is worth pointing out that it is very easy to show that some programs terminate (consider the earlier program for computing the product). But, as we will see, it is not possible *in general* to do this i.e. for any program.

**Which Program is Best?**

It should not be surprising that it is possible to write two programs that compute the same function. As a simple case consider `int x =1; int y=2` and `int y=2; int x=1`. Syntactically these programs are different but they compute the same function.

Less abstractly, let’s consider the following two methods that find the maximum element in an array. They are both correct i.e. they both find the maximum value. So which program is best?

```java
public int max(int[] a) {
    int max = a[0];
    int len = a.length;
    for (int i=1; i<len; i++){
        if (a[i]>max){
            max = a[i];
        }
    }
    return max;
}
```

```java
public int max(int[] a) {
    int len = a.length;
    for (int i=0; i<len; i++){
        boolean isMax=true;
        for (int j=0; j<len; j++){
            if (a[i]<a[j]){
                isMax=false;
            }
        }
        if (isMax){
            return a[i];
        }
    }
    // unreachable
    return 0;
}
```
Well, the one on the right is longer, does that mean it is best? Or perhaps the one on the left is easier to read so that is best. Okay, it seems that we need to define what qualities we want in a program before we can answer this question. Typically computer scientists and programmers have a keen (sometimes unhealthy) interest in how fast something is. So let us rephrase the question, which one is faster?

If you said the one on the left you are right, but do you understand why? Hint: it’s not because the one on the right looks more complicated. Let’s try running them with different length arrays and see what happens.

The following table tabulates the running times on my laptop for the two different programs on different length arrays of random integers (the contents doesn’t matter as we have to look at every element).

<table>
<thead>
<tr>
<th>Array length</th>
<th>Time in milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left \textit{max}</td>
</tr>
<tr>
<td>1,000</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0</td>
</tr>
<tr>
<td>10,000,000</td>
<td>4</td>
</tr>
</tbody>
</table>

The left \textit{max} mostly says 0 because it took less than 1 millisecond, and that’s the shortest amount of time \texttt{java} will accurately let us measure. However, the numbers for the right \textit{max} are pretty big, which is bad, so what’s going on. The left \texttt{max} method performs \(n\) checks where \(n\) is the length of the list. It always performs exactly \(n\) checks as it needs to look at every element in the array. The right \texttt{max} method performs \(n\) checks when the first element is the maximum already. But if the last element is the maximum it will perform \(n^2\) checks. Try typing \((1,000,000)^2\) into your calculator - that’s a big number. This demonstrates that it is easy to write a program that solves the same problem and make it very slow\(^3\).

We call \(n\) and \(n^2\) the \textit{worst-case asymptotic complexity} of each method respectively. This discussion has left out a few important points but should give a flavour of what complexity analysis is about. We discuss this further in Chapter 3.

\(^3\)Note that it is probably not just time complexity that is making the right \texttt{max} slow. The left \texttt{max} looks at each element of the array once, so it is very friendly for memory cache accesses. However, for large arrays, the right \texttt{max} method will need to load the whole array into cache multiple times.
What can we compute

We now turn to a series of somewhat more abstract questions. Now that we have a model of computation, we can use it to describe the functions that we are able to compute. This is straightforward. We can compute a function if it is possible to write a while program to compute it. Note that the above argument that all functions of interest can be mapped to functions in $\mathbb{N} \to \mathbb{N}$ means that this argument applies to all computable functions.

What can we not compute

The above description of what a computable function suggests a very easy answer: we cannot compute functions for which we cannot write a while program. However, it is not immediately obvious that this set is nonempty. Here (and later) we show that there exist uncomputable functions by showing that there must be more functions than programs and giving an example of an uncomputable function.

Counting Functions and Programs

Firstly, we need to quickly remind ourselves what countable and uncountable sets are. A set is countable if there is a bijection between it and $\mathbb{N}$ (if you can’t remember what a bijection is then I suggest looking it up at this point). A set is uncountable if it is not countable!

Now we argue that the set of programs is countable. If we are lazy we can argue that any program is a finite sequence of symbols and we can code any finite sequence as a natural number. If we are more imaginative then we can show how to map language constructs directly to numbers. In both approaches we take a program and assign a unique number to that program (this is called Gödel numbering). We can use this mapping to give a bijection between programs and the natural numbers, hence programs are countable.

Next we argue that the set of functions $\mathbb{N} \to \mathbb{N}$ is uncountable. We can do this via Cantor’s diagonal argument. The details are given in Section 4.2 but the general idea is that given any enumeration of functions we can create a function not in that enumeration. Put alternatively, if you try counting functions in $\mathbb{N} \to \mathbb{N}$ we can prove that you will always miss one.

It should be obvious that if one set is countable and the other uncountable then they cannot be the same size, therefore there must be more functions than programs, and therefore functions for which no program exists. However, this proof does not give us an example of an uncomputable function, only show us that one must exist.
An Uncomputable Function: The Halting Problem

So we have shown that uncomputable functions must exist but do we really believe that without seeing one? We should, because we proved it, but that’s not as satisfying as seeing one. Section 4.4 gives a detailed introduction to a famous uncomputable function captured by the Halting Problem. As this is quite involved I don’t repeat the description here. The general result is that it is not possible to write a program that, given any program as input, decides whether that program halts.

Computational Universality

There are various models of computation, with Turing Machines generally agreed to be the de facto standard definition. Other models of computation are then said to be Turing-Complete if they can be used to model Turing Machines. Later we will give a high level sketch of how our while language can model Turing Machines, thus showing that the language is Turing-Complete. Indeed, there is a broader idea that any sensible model of computation can be used to model Turing Machines, and thus all such models of computation compute the same functions, this is called the Church-Turing thesis.

At the root of the Turing-Completeness argument is a notion of universality. This is the idea that a model of computation is universal if it can model itself i.e. can be used to write a program that can execute any program that can be written in that model. If this seems odd then think about C compilers written in C, this is what we mean.

To demonstrate universality we will use a Universal Program that takes an encoded program and its input and runs the program on the input. Note that this relies on our previous results that we can encode programs as numbers. We don’t even need to write out this Universal Program; we just need to be happy that it exists. For it to exist we just need a computable semantics (we have this) and a method for encoding/decoding programs (we have this).

Summary

This introduction has briefly visited all of the core concepts in this Part of the course. You still need to read the rest of the notes but hopefully this introduction serves as a map, connecting the different concepts together and giving a slightly more concise and informal introduction to the ideas.

As with any part of these notes, if anything is unclear or could be improved in general then please let me know.
Contents

0 Recap 3
  0.1 Functions ................................................. 3
  0.2 Countability .............................................. 4

1 The while Programming Language 5
  1.1 What is a programming language? .................... 6
  1.2 Defining the syntax of while ......................... 6
  1.3 Defining the Execution of while Programs .......... 13
  1.4 Coding Data Structures ................................. 20
  1.5 Adding Arrays ............................................ 25

2 Proving while Programs Correct 33
  2.1 What does Correctness Mean? ......................... 34
  2.2 Proving the Division Program is Correct ............ 37
  2.3 An Axiomatic System for Partial Correctness ....... 41
  2.4 Some More Hints ........................................... 57
  2.5 An Axiomatic System for Total Correctness .......... 61
  2.6 What about Arrays? ...................................... 64
  2.7 Practical Correctness .................................... 64
Chapter 0

Recap

This chapter recaps some of the definitions and concepts you should already know (from COMP11120) that will be used in this course. This material will not be covered in lectures in any great detail and is here to make these notes self-contained. It will not be explicitly examined but its contents will be assumed for the exam, as they are prerequisite for the rest of the course.

0.1 Functions

In Section 2.5 of the COMP11120 notes you met some properties of functions. Notably, injectivity, surjectivity and bijectivity. There the treatment was relatively informal, here I introduce the same concepts using formal logic definitions. Remember that we write \( \phi : \alpha \rightarrow \beta \) as a function with domain \( \alpha \) and co-domain \( \beta \). We can also say that \( \alpha \rightarrow \beta \) is the type of \( \phi \).

Definition 0.1.1

A function \( \phi : \alpha \rightarrow \beta \) is injective if

\[
\forall (x, y : \alpha). f(x) = f(y) \Rightarrow x = y
\]

A function \( \phi : \alpha \rightarrow \beta \) is surjective if

\[
\forall (b : \beta). \exists (x : \alpha). f(x) = b
\]

Definition 0.1.2

A bijection \( \phi : \alpha \rightarrow \beta \) is a function from \( \alpha \) to \( \beta \) which is both injective and surjective.
We also want the notion of an inverse of a function.

**Definition 0.1.3**

We say that for function $f : \alpha \to \beta$ the function $f^{-1} : \alpha \to \beta$ is the inverse of $f$ if

$$\forall (x : \alpha) : f(f^{-1}(x)) = x$$

You should recall that not all functions have an inverse (e.g. $f(x) = 0$). We can use the notion of inverse function to give an alternative definition of bijection.

**Definition 0.1.4**

A bijection $\phi : \alpha \to \beta$ is a function from $\alpha$ to $\beta$ which has an inverse.

At this point it would be normal to prove that our two definitions of bijection are equivalent, but as this is a recap I won’t do this. The reading list contains books that contain such proofs.

### 0.2 Countability

We will also depend on some definitions about the cardinality of sets. This was covered in Section 5.2 of COMP11120 but I give a different (but equivalent) definition of these concepts.

**Definition 0.2.1**

A set $S$ is countably infinite if, and only, if there exists a bijection $\phi : S \to \mathbb{N}$.

I assume that you are familiar with the set $\mathbb{N}$ of natural numbers. If not, please remind yourself as it will be important in this part of the course.

**Definition 0.2.2**

A set $S$ is countable if, and only if, it is finite or it is countably infinite.

**Definition 0.2.3**

A set $S$ is uncountable if, and only if, it is not countable.
Chapter 1

The while Programming Language

As argued in the introduction to this part, in order to discuss computation we need a simple programming language (to provide a model of computation). As we will be discussing the formal semantics of this language, and later providing an axiomatic system, we need it to be small, and simple. Unfortunately most real programming languages are far too large and complicated. As we shall see, a larger language can be much easier to use.

Learning Outcomes

At the end of this Chapter you will:

• Be familiar with the simple imperative programming language while;
• Be able to write simple programs in while;
• Understand the notion of state and formal semantics and be able to use these to show how a while program executes on a given state
• Understand how data structures (such as pairs, or syntax trees) can be coded as natural numbers; and
• Understand how functions on data structures become arithmetic operations;

Additionally, you may also understand how the while language can be extended with arrays but this part of the Chapter is not examinable, and exists only so that we can have a more in-depth discussion in Chapter 3.
1.1 What is a programming language?

Before we define the while programming language let us briefly consider what a programming language is. In simple terms, it is a language we use to describe to a computer what it should do. To be unambiguous it is necessary, therefore, that a programming language has a well-defined (formal) syntax describing the form of valid programs. However, this is not sufficient for communicating what a computer should do, it is also necessary to know what programs written in a programming language mean. To achieve these we must describe the language’s semantics. This can be done at varying levels of formality; in this Chapter we provide a small-step structural operational semantics for while, which is really rather formal. Many real programming languages have a well-defined syntax. Fewer have a well-defined semantics. Very few have a formal semantics. One of the aims of this chapter is to give you a flavour of programming language design and the associated formal concepts.

1.2 Defining the syntax of while

The first thing we will do is define the grammar of our tiny programming language (if this looks unfamiliar then revisit Chapter 4 of Part A of this course):

\[
S ::= x := a | \text{skip} | S_1; S_2 | \text{if } b \text{ then } S_1 \text{ else } S_2 | \text{while } b \text{ do } S
\]

\[
b ::= \text{true} | \text{false} | a_1 = a_2 | a_1 \leq a_2 | \neg b | b_1 \land b_2
\]

\[
a ::= x | n | a_1 + a_2 | a_1 - a_2 | a_1 \times a_2
\]

The variables S, b and a are non-terminals of our grammar and represent (respectively) Statements (Stm), Boolean Expressions (BExp), and Arithmetic Expressions (AExp). You will also notice we have variables in the programming language which we’ve represented by x, and numerals represented by n. Any of these symbols can be dashed (x’) or subscripted (S_1). The numerals n can be thought of as strings of digits.

Once we have a grammar the first thing we want to do is use it to write a program. Here is our first while program:

\[
x := 1; \ y := 5
\]
\[
\text{while } (y > 0)(x := x + x; \ y := y - 1;)
\]

Have a think about what this computes. We will return to it in a few pages and discuss it further. For now just make sure you are comfortable that this program can be generated by the above grammar.
1.2. DEFINING THE SYNTAX OF WHILE

Let us note that our grammar is ambiguous. This means that when we write
\[ x := 17; \textbf{while} (0 \leq x) \textbf{do} x := x-1; \quad x := x+1 \]
we could mean one of two programs. One choice is:
\[ x := 17; \textbf{while} (0 \leq x) \textbf{do} (x := x-1; \quad x := x+1) \]
But what we probably meant was instead:
\[ x := 17; (\textbf{while} (0 \leq x) \textbf{do} x := x-1); \quad x := x+1 \]
To disambiguate the expressions and sentences we use the parentheses ‘(’ and ‘)’. Note that the use of the brackets for statements is not usual in programming languages, but very common in mathematics.

Exercise 1.1

Why is the grammar for \textbf{while} ambiguous? Give one example of an ambiguous Arithmetic Expression, Boolean Expression, and Statement. You may wish to revisit Section 4.4 of Part A.

1.2.1 Extending the Language

The \textbf{while} language is missing a lot of the things we would normally want to write. However, it contains the building blocks necessary to define these things. For example, one might want to write the statement
\[ \textbf{if} b \textbf{then} S \]
\[ \textit{i.e.} \] without an \textbf{else} branch. But we only have \textbf{if} b \textbf{then} \textit{S}_1 \textbf{else} \textit{S}_2 in the \textbf{while} language. To be able to capture the required statement we need to write an equivalent \textit{legal} \textbf{while} program. In this case we can note that
\[ [\textbf{if} b \textbf{then} S] \equiv [\textbf{if} b \textbf{then} S \textbf{else} \textbf{skip}] \]
This says that the lefthand side (if-then) is equivalent to the righthand side (if-then-else-skip), for all boolean expressions \( b \) and statements \( S \).

Whenever we find it helpful, we may mark off parts of the programming language from the other mathematics which we use to describe it; for this purpose we will use \textit{semantic brackets} \([S]\).

Exercise 1.2

How can we represent the boolean expression \( b_1 \lor b_2 \) in the \textbf{while} language? (Hint: recall DeMorgan’s laws)
Exercise 1.3

How can we represent the boolean expression \( b_1 \rightarrow b_2 \) in the \texttt{while} language?

Exercise 1.4

How can we represent the statement

\begin{verbatim}
do S while b
\end{verbatim}

in the \texttt{while} language? Note that the intended semantics is that \( S \) is executed at least once and then while \( b \) is true.

In these notes we may use shorthands such as \texttt{if} \( b \) \texttt{then} \( S \) where their definition as equivalent programs or expressions in the \texttt{while} language would be obvious. Why don’t we simply add these to the language? As mentioned previously, we want the language to be small so that it is easier to define and reason about. It is common to define a small set of statements and expressions in which more interesting statements and expressions can be defined.

Exercise 1.5

Name three features not already mentioned, which are missing from the \texttt{while} programming language. Try to name the most significant omissions, rather than variants of the previous exercises.

1.2.2 Writing Programs in \texttt{while}

We have a programming language so let us write some programs. As a first example consider the following program:

\begin{verbatim}
x:=1; y:=5
while (y>0)(x:=x+x; y:=y-1;)
\end{verbatim}

What does it do? Pause here and think about it. If it is not obvious go to a computer and type

\begin{verbatim}
x=1; y=5
while y>0:
x=x+x
y=y-1
print x,y
\end{verbatim}
1.2. DEFINING THE SYNTAX OF WHILE

into python\(^1\) and see what happens. Notice how similar our while program
looks to the python. At the end of this program the variable \(x\) will contain
\(2^5 = 32\) and we will say that this is what the program computes.

What if I want to compute \(2^n\) for general \(n\)? I need input. We will gener-
ally assume that variables can contain values before executing the program
and that we can read values out of variables at the end of the program. This
is how input and output worked for the first computers.

Modifying the above program is straightforward. Assuming that variable
\(y\) contains the value \(n\) the following program computes \(2^n\):

\[
x := 1;\\
\text{while}(y > 0)(x := x + x; \ y := y - 1;)
\]

Now \(x\) will contain \(2^y\) at the end of the program.

The following five examples show how to write more complex programs
in the while programming language. I strongly encourage you to treat
these as exercises and attempt to write the corresponding programs before
looking at the answers. The solutions have been included to expose you to
a reasonable number of while programs and to give us programs to discuss
later in these notes.

Example 1.2.1

Show how to write a division program. You are to accept the numer-
ator and divisor in the variables \(x\) and \(y\) respectively, and to deliver
the result of the division and the remainder in \(d\) and \(r\) respectively.
You may assume that \(x, y \in \mathbb{N} \land y \neq 0\).

Example 1.2.2

Write an integer square root program, i.e. a program which accepts
an argument passed in in variable \(x \in \mathbb{N}\), and calculates \(z = [\sqrt{x}]\),
and \(r = x - z^2\), the remainder.

Example 1.2.3

Write a program to compute the \(n\)th Fibonacci number. Assume that
variable \(x \in \mathbb{N}\) contains \(n\) and place the result in variable \(z\).

\(^1\) Just type python on the command line to get the interactive mode, at least on any School machine.
CHAPTER 1. THE WHILE PROGRAMMING LANGUAGE

Example 1.2.4

Write a program to compute the greatest common divisor (gcd) between two numbers i.e. a program which accepts arguments passed in variables \( x \in \mathbb{N} \) and \( y \in \mathbb{N} \), and places in \( z \) the largest number that divides both \( x \) and \( y \) with no remainder. For further clarification of gcd you may wish to refer to the Wikipedia page.

Example 1.2.5

Write a program to decide whether a number is a prime number. Assume that variable \( x \in \mathbb{N} \) contains the input variable and place 1 in \( z \) if \( x \) is prime and place 0 in \( z \) otherwise.

Answer to Example 1.2.1

A division program can be written as follows:

```plaintext
r := x;
d := 0;
while y \leq r do (d := d+1; r := r-y)
```

To see why how works consider \( x = 10, y = 3 \) and write down \( r \) and \( d \) for each iteration of the loop:

<table>
<thead>
<tr>
<th>( r )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

and we get the answer \( r = 1, d = 3 \) as expected. In simple terms, we are counting the number of times we can remove \( y \) from \( x \).

Answer to Example 1.2.2

There are lots of very clever ways to perform this calculation, but we’ll choose the very simplest.

```plaintext
z := 0;
while ((z+1)*(z+1) \leq x) do z := z+1;
\( r := x - z*z \)
```
1.2. DEFINING THE SYNTAX OF WHILE

In effect we start with an initial guess that the square root is 0 and increment the guess at each step until a number is found that is “too big”. If this program is confusing then do what we did before and write out the table of intermediate values.

Answer to Example 1.2.3

This is a standard programming task but here we do not have access to many of the language features we may be used to.

\[\begin{align*}
y &:= 1; \\
z &:= 0; \\
\text{while } (x > 0) \text{ do } ( \\
& \quad t := z; \\
& \quad z := y; \\
& \quad y := t + y; \\
& \quad x := x - 1
\end{align*}\]

Here we use \(z\) to remember the \(n-1\)th value in the sequence.

Answer to Example 1.2.4

This can be solved using Euclid’s algorithm as follows.

\[\begin{align*}
\text{while } (x \neq y) \text{ do } ( \\
& \quad \text{if } (x > y) \text{ then} \\
& \quad & \quad x := x - y \\
& \quad \quad \text{else} \\
& \quad & \quad y := y - x \\
& \quad ) \\
& \quad z := x
\end{align*}\]

Note that this program is destructive i.e. it modifies its input. As an exercise you might want to try rewriting it to be nondestructive.

Answer to Example 1.2.5
Testing for primality is straightforward - we just check each candidate for a divisor. Normally we would start this process at the square root of \( x \) (as no factor can be greater than this) but we have kept the program as simple as possible.

\[
y := x; \\
z := 0; \\
\text{while } (y > 1 \land z = 0) \text{ do } ( \\
    r := x; \\
    (\text{while } (y \leq r) \text{ do } r := r - y); \\
    (\text{if } (r = 0) \text{ then } z := 1); \\
    y := y - 1; \\
) \\
\]

This program shows us that it would be useful if we could structure our programs so that we could refer to subprograms (note that we perform division here). We will be returning to this program when we discuss complexity. Note that this is the only program we have seen so far that contains a nested while loop.

\[\square\]

**Important**

Remember to answer these exercises using the \texttt{while} language as defined in this section. It is allowed to use derived statements as long as you can show how they can be defined in terms of the available constructs.

**Exercise 1.6**

Extend the program given in Example 1.2.1 so that it works for general integers e.g. it handles the case where \( x \) and \( y \) can be negative.

**Exercise 1.7**

Write a program to calculate whether a number is even and another program to calculate whether a number is odd. Now write a program to compute \( x \mod y \).
1.3. **DEFINING THE EXECUTION OF WHILE PROGRAMS**

The previous section introduced the syntax of the while programming language and we discussed how to write programs in it but we only have an informal or intuitive sense of what the programs mean. To fully describe a programming language we also need to capture the semantics.

**Exercise 1.8**

Write a program to calculate the factorial of non-negative integer in variable $x$ returning the result in variable $r$.

**Exercise 1.9**

Write a power program, i.e. a program which accepts two arguments passed in in variables $x$ and $y$, and calculates $x^y$, placing the value in variable $r$. You may assume that both $x, y \in \mathbb{N}$. What answer do you propose to give to $0^0$? **Hint:** Consider the two sequences \{0, 1, 2, \ldots\} and \{0, 1, 2^0\}.

**Exercise 1.10**

Extend the power program from Exercise 1.9 to handle negative $x$ and $y$.

**Exercise 1.11**

Write a logarithm-base2 program, i.e. a program which accepts an argument passed in variable $x \in \mathbb{N}$, and calculates $z$ and $r$ such that $x = 2^z + r$ where $0 \leq r < 2^z$. Assume that $x \neq 0$. **Hint:** You may find it useful to refer to the division program we saw earlier.

**Exercise 1.12**

Extend your answer to Exercise 1.11 to take an arbitrary base in variable $b$.

**Exercise 1.13**

Write a program to calculate the least common multiple of two positive integers in variables $x$ and $y$ returning the result in variable $r$. 

1.3 **Defining the Execution of while Programs**
1.3.1 The Notion of State

The important feature of an imperative programming language, such as while, is that it has a state in which the current values of the variables are stored. There are several alternative ways to think about the state. One is as a table

\[
\begin{array}{|c|}
\hline
x & 5 \\
\hline
y & 7 \\
\hline
z & 0 \\
\hline
\end{array}
\]

or as a “list” of the form

\[ [x \mapsto 5, \ y \mapsto 7, \ z \mapsto 0] \]

In all cases we must ensure that there is only one value associated with each variable. By requiring the state to be a function this requirement is trivially fulfilled, whereas for the alternatives extra conditions have to be enforced. We will use total functions to model this because we want only one value for each variable. The total functions for States take a variable to an integer.

**Definition 1.3.1**

The type State is a total function

\[
\text{State} = \text{Var} \to \mathbb{Z}
\]

Confused?

What is this State thing? Formally it is a function space i.e. a set of functions that have the same domain (here Var) and co-domain (here \( \mathbb{Z} \)). Informally, we can think of states as bits of memory that are infinitely indexed and hold numbers.

We will assume that any variable not mentioned when we set up a state is 0. This avoids us having to worry about which variables are defined.

We will also use the following notation to describe states:

\[ [x \mapsto 5, \ y \mapsto 7, \ z \mapsto 0] \]

This is the function which, when applied to \( x \) returns 5, when applied to \( y \) returns 7, and when applied to anything else returns 0.
1.3. DEFINING THE EXECUTION OF WHILE PROGRAMS

Suppose that \( s = [x \mapsto 5, \ y \mapsto 7, \ z \mapsto 0] \). Then applying the function \( s \) to variable \( x \) gives 5; we write this as:

\[ s(x) = 5 \]

With this notation we can now define a recursive function \( A \) to determine the value represented by an arithmetic expression using Table 1.3. We do something similar with boolean expressions using Table 1.4, except that they return a truth value in \( \mathbb{B} \), which is either \( \texttt{tt} \) or \( \texttt{ff} \) (true or false, respectively). Don’t worry too much about these tables, they capture the obvious semantics of arithmetic and boolean expressions. Understanding the transition system in the next section is more important.

Apart from looking up the values of variables in states, we must also be prepared to change the state when we do an assignment. For example, if we do the assignment \( x := 3 \) and we currently have the state \( s \), then the new state \( s' \) will return 3 when \( s' \) is applied to \( x \), and otherwise behaves exactly as \( s \) does on every other variable.

**Definition 1.3.2**

We define the state modification notation – using square brackets – as follows. Let \( s' = s[x \mapsto v] \); then

\[ s'(x) = v \]

And whenever \( y \neq x \):

\[ s'(y) = s(y) \]

Thus \( s' \) is the same as \( s \) except when it is applied to the variable \( x \), in which case it returns the new value of \( x \) (which is \( v \)) rather than the old one.

**Example 1.3.3**

Suppose that the state \( s = [x \mapsto 3, \ y \mapsto 6] \). Then every value apart from \( x \) and \( y \) will be given the value 0.

The modified state \( s' = s[x \mapsto 17] \), which we obtain after executing the assignment:

\[ x := 17 \]

is: \( s' = [x \mapsto 17, \ y \mapsto 6] \).

If it helps, what we are doing when we write \( s[x \mapsto v] \) is composing two functions i.e. producing a new function \( s \circ [x \mapsto v] \).
\[ A : \mathbf{AExp} \rightarrow \text{State} \rightarrow \mathbb{Z} \]
\[ A[n] s = N[n] \]
\[ A[x] s = s(x) \]
\[ A[a_1 + a_2] s = A[a_1] s + A[a_2] s \]
\[ A[a_1 \ast a_2] s = A[a_1] s \ast A[a_2] s \]
\[ A[a_1 - a_2] s = A[a_1] s - A[a_2] s \]

Table 1.3: Defining \( A \) for Arithmetic Expressions

\[ B : \mathbf{BExp} \rightarrow \text{State} \rightarrow \mathbb{B} \]
\[ B[\text{true}] s = \text{tt} \]
\[ B[\text{false}] s = \text{ff} \]
\[ B[a_1 = a_2] s = \begin{cases} 
\text{tt} & \text{if } A[a_1] s = A[a_2] s \\
\text{ff} & \text{if } A[a_1] s \neq A[a_2] s 
\end{cases} \]
\[ B[a_1 \leq a_2] s = \begin{cases} 
\text{tt} & \text{if } A[a_1] s \leq A[a_2] s \\
\text{ff} & \text{if } A[a_1] s > A[a_2] s 
\end{cases} \]
\[ B[\neg b] s = \begin{cases} 
\text{tt} & \text{if } B[b] s = \text{ff} \\
\text{ff} & \text{if } B[b] s = \text{tt} 
\end{cases} \]
\[ B[b_1 \land b_2] s = \begin{cases} 
\text{tt} & \text{if } B[b_1] s \text{ and } B[b_2] s \\
\text{ff} & \text{if } \text{not} (B[b_1] s \text{ and } B[b_2] s) 
\end{cases} \]

Table 1.4: Defining \( B \) for Boolean Expressions

\[ \text{[ass]} \quad < x := a, s > \Rightarrow s[x \mapsto A[a]] s \]
\[ \text{[skip]} \quad < \text{skip}, s > \Rightarrow s \]

\[ \begin{align*}
\text{[comp\textsuperscript{1}]} & \quad < S_1, s > \Rightarrow < S'_1, s' > \quad < S_1; S_2, s > \Rightarrow < S'_1; S_2, s' > \\
\text{[comp\textsuperscript{2}]} & \quad < S_1, s > \Rightarrow s' \quad < S_1; S_2, s > \Rightarrow < S_2, s' >
\end{align*} \]

\[ \text{[if\textsuperscript{1}]} \quad < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_1, s > \text{ if } B[b] s = \text{tt} \]
\[ \text{[if\textsuperscript{2}]} \quad < \text{if } b \text{ then } S_1 \text{ else } S_2, s > \Rightarrow < S_2, s > \text{ if } B[b] s = \text{ff} \]
\[ \text{[while]} \quad < \text{while } b \text{ do } S, s > \Rightarrow \\
\begin{align*}
& < \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s > 
\end{align*} \]

Table 1.5: The \textbf{While} Language
1.3. DEFINING THE EXECUTION OF WHILE PROGRAMS

1.3.2 A Transition System for Statements

In Table 1.5 we have defined the sequence of transitions required to execute a while program. This is merely a more complicated state transition system than the ones associated with regular expressions. Indeed there is a clue in the naming: for regular expressions we use finite state automata. We do not use that name here, because the transition system presented in Table 1.5 has an infinite number of possible statements $S$.

The rules operate on a configuration $<S, s>$. The first component of $<S, s>$ is a while program, the second is a state. In making a transition, we are executing the first statement within $S$, and perhaps making changes to the state. The $\Rightarrow$ symbol is not being used as implication in the table, instead it is defining a relation between either pairs of tuples, or a tuple and a final state. Of course we read the $\Rightarrow$ symbol as if the program were making progress towards some final state.

Some transitions are defined so that they only apply when a boolean condition is met (the [if$^c$] case in the table). Others (e.g. [comp$^2$]) are defined by way of a horizontal line. This line is to be read as: “If the transition above the line is possible, then the transition below the line is also possible.”

The transition system given in Table 1.5 – along with Tables 1.3 and 1.4 – is an example of a formal semantics for a programming language. Another (less pretentious) word for semantics is “meaning”.

Example 1.3.4

Let us look at an example execution:

$$<\text{if } (0 \leq x) \text{ then skip else } x := 0 - x, [x \rightarrow -17]> \Rightarrow <x := 0 - x, [x \rightarrow -17]> \quad \text{Rule [if$^c$]}$$

$$\Rightarrow [x \rightarrow 17] \quad \text{Rule [ass]}$$

What is the effect of this program in general? **Hint:** what can you say about the value of $x$ after the execution has finished?

Example 1.3.5

Let us look at another example execution:

$$<x := 0; \text{while } (x > 0) \ (x := x - 1), []> \Rightarrow <\text{while } (x > 0) \ (x := x - 1), [x \rightarrow 0]> \quad \text{Rule [ass]}$$
What just happened? We had something of the form $S_1; S_2$ and now we have $S_2$ so we must have applied $\text{[comp]}^2$. Here we are cheating and only writing the rule that took $S_1$ and produced an updated state, leaving the application of $\text{[comp]}$ implicit. In general, the application of either $\text{[comp]}$ rules will be implicit. Now let’s continue.

$\Rightarrow \langle \text{if} \ (x > 0) \ \text{then} \ (x := x - 1; \ \text{while} \ (x > 0) \ (x := x - 1)) \ \text{else} \ \text{skip}, [x \mapsto 0] \rangle$ \hspace{1cm} \text{Rule [while]}

$\Rightarrow \langle \text{skip}, [x \mapsto 0] \rangle$ \hspace{1cm} \text{Rule [if$^0$]}

$\Rightarrow [x \mapsto 0]$ \hspace{1cm} \text{Rule [skip]}

**Example 1.3.6**

Now let us take a look at a larger example using the following program for computing $2^n$ defined earlier:

```plaintext
x := 1;
while (y > 0) (x := x + x; y := y - 1;)
```

This program requires that $n$ is loaded into $y$ before we begin so let us consider the starting state $[y \mapsto 5]$. We then apply the semantic rules as follows:

$\langle x := 1; \text{while} \ (y > 0) \ (x := x + x; y := y - 1), [x \mapsto 1, y \mapsto 5] \rangle$ \hspace{1cm} \text{Rule [ass]}

At this point we realise that expanding the program is going to involve lots of writing so we start introducing some *definitions*. Let

$$P \triangleq \text{while} \ (y > 0) \ (x := x + x; y := y - 1)$$

in the following$^2$. Now let’s continue

$\Rightarrow \langle \text{if} \ (y > 0) \ \text{then} \ (x := x + x;
\hspace{0.5cm} y := y - 1; \ P) \ \text{else} \ \text{skip}, [x \mapsto 1, y \mapsto 5] \rangle$ \hspace{1cm} \text{Rule [while]}

$\Rightarrow \langle x := x + x; y := y - 1; P; [x \mapsto 1, y \mapsto 5] \rangle$ \hspace{1cm} \text{Rule [if$^1$]}

$\Rightarrow \langle y := y - 1; P; [x \mapsto 2, y \mapsto 5] \rangle$ \hspace{1cm} \text{Rule [ass]}

$\Rightarrow \langle P; [x \mapsto 2, y \mapsto 4] \rangle$ \hspace{1cm} \text{Rule [ass]}

---

$^2$I use $\triangleq$ in these notes when I define something to be equal to something else. Such equalities only exist to introduce names for things.
1.3. DEFINING THE EXECUTION OF WHILE PROGRAMS

At this point we can tell that we are going to repeat ourselves as we have returned to the program we have already seen, this will happen when executing loops. Let’s finish this.

\[
\begin{align*}
\Rightarrow & \quad < \mathbf{if} \ (y > 0) \ \mathbf{then} \ (x := x + x; \\
& \quad y := y - 1; P) \ \mathbf{else} \ \mathbf{skip}, [x \mapsto 2, y \mapsto 4] > \quad \text{Rule [while]} \\
\Rightarrow & \quad < x := x + x; y := y - 1; P, [x \mapsto 2, y \mapsto 4] > \quad \text{Rule [if*]} \\
\Rightarrow & \quad < y := y - 1; P, [x \mapsto 4, y \mapsto 4] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < P, [x \mapsto 4, y \mapsto 3] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < \mathbf{if} \ (y > 0) \ \mathbf{then} \ (x := x + x; \\
& \quad y := y - 1; P) \ \mathbf{else} \ \mathbf{skip}, [x \mapsto 4, y \mapsto 2] > \quad \text{Rule [while]} \\
\Rightarrow & \quad < x := x + x; y := y - 1; P, [x \mapsto 4, y \mapsto 2] > \quad \text{Rule [if*]} \\
\Rightarrow & \quad < y := y - 1; P, [x \mapsto 8, y \mapsto 3] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < P, [x \mapsto 8, y \mapsto 2] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < \mathbf{if} \ (y > 0) \ \mathbf{then} \ (x := x + x; \\
& \quad y := y - 1; P) \ \mathbf{else} \ \mathbf{skip}, [x \mapsto 8, y \mapsto 2] > \quad \text{Rule [while]} \\
\Rightarrow & \quad < x := x + x; y := y - 1; P, [x \mapsto 8, y \mapsto 2] > \quad \text{Rule [if*]} \\
\Rightarrow & \quad < y := y - 1; P, [x \mapsto 16, y \mapsto 2] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < P, [x \mapsto 16, y \mapsto 1] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < \mathbf{if} \ (y > 0) \ \mathbf{then} \ (x := x + x; \\
& \quad y := y - 1; P) \ \mathbf{else} \ \mathbf{skip}, [x \mapsto 16, y \mapsto 1] > \quad \text{Rule [while]} \\
\Rightarrow & \quad < x := x + x; y := y - 1; P, [x \mapsto 16, y \mapsto 1] > \quad \text{Rule [if*]} \\
\Rightarrow & \quad < y := y - 1; P, [x \mapsto 32, y \mapsto 1] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < P, [x \mapsto 32, y \mapsto 0] > \quad \text{Rule [ass]} \\
\Rightarrow & \quad < \mathbf{if} \ (y > 0) \ \mathbf{then} \ (x := x + x; \\
& \quad y := y - 1; P) \ \mathbf{else} \ \mathbf{skip}, [x \mapsto 32, y \mapsto 0] > \quad \text{Rule [while]} \\
\Rightarrow & \quad < \mathbf{skip}, [x \mapsto 32, y \mapsto 0] > \quad \text{Rule [if*]} \\
\Rightarrow & \quad [x \mapsto 32, y \mapsto 0] \quad \text{Rule [skip]}
\end{align*}
\]

Exercise 1.14

What happens when you execute the following program:

\[\text{while true do skip}\]

**Hint:** Think about what this program does before expanding the transitions!

Exercise 1.15

Apply the semantic rules to the state \([x \mapsto 8, y \mapsto 3]\) for the division program given in Example 1.2.1. Now try applying them to the state \([x \mapsto 5, y \mapsto -2]\), what happens?
Exercise 1.16

Apply the semantic rules to the state \([x \mapsto 5, y \mapsto -2]\) for the extended division program you wrote in Exercise 1.6.

Exercise 1.17

Apply the semantic rules to the states \([x \mapsto 9]\) and \([x \mapsto 10]\) for the square root program given in Example 1.2.2.

Exercise 1.18

Apply the semantic rules to the state \([x \mapsto 5]\) for the Fibonacci program given in Example 1.2.3.

Exercise 1.19

Apply the semantic rules to the state \([x \mapsto 9, y \mapsto 3]\) for the gcd program given in Example 1.2.4.

Exercise 1.20

Apply the semantic rules to the states \([x \mapsto 3]\) and \([x \mapsto 4]\) for the primality testing program given in Example 1.2.5.

(*) Exercise 1.21

Using your favourite programming language to write an interpreter for the while language. The semantic definitions given in this section should be sufficient. If you are feeling very ambitious you could also write a parser. Otherwise, introduce an API that allows you to construct while programs programatically to be passed to your interpreter. Anybody who does this and lets me have the code to distribute in future years gets a large bar of chocolate (only available to the first solution in each programming language, modulo race conditions).

1.4 Coding Data Structures

With the definitions given so far, you could now implement this language, although this is not the purpose of this course. Of much more importance is that you understand how to encode data structures as natural numbers. By showing that arbitrary data structures can be coded as natural numbers we convince ourselves that we only need to consider programs on natural numbers.
1.4. CODING DATA STRUCTURES

1.4.1 Pairs

In this section we will introduce a function that can be used to encode and decode pairs of natural numbers as a single natural number i.e. a bijective function in \((\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}\). See page 3 for a recap of what a bijection is (it’s important in this section).

The function we introduce is

\[
\phi(n, m) = 2^n(2m + 1) - 1
\]

which takes any pair of natural numbers and produces a unique natural number. Before we prove that this function is bijective let’s consider what it is doing. The following table gives its value for some different values of \(n\) and \(m\).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>…</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>19</td>
<td>…</td>
</tr>
<tr>
<td>3</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

If we consider what these numbers would look like written in binary we will see that this produces \(m\) in binary, followed by 0 and then \(n\) 1s. For example, \(\phi(2, 3) = 27\), which is 11011 in binary i.e. 11 followed by 011. This example should already make us feel quite confident that \(\phi\) is a bijection as we have in our heads an algorithm that could take a binary number and get \(n\) and \(m\) back.

**Theorem 1.4.1**

The function

\[
\phi : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}
\]

\[
\phi(n, m) = 2^n(2m + 1) - 1
\]

is bijective.

We will show this in two stages.

**Lemma 1.4.2**

\(\phi\) is an injection, i.e.

\[
\forall(n, m, n', m' \in \mathbb{N}) : \phi(n, m) = \phi(n', m') \Rightarrow (n = n' \land m = m')
\]
Proof of 1.4.2

Suppose that \(2^n(2m+1) - 1 = \phi(n, m) = \phi(n', m') = 2^{n'}(2m' + 1) - 1\).
Eliminating the \(-1\) from each side, we can assume that:

\[2^n(2m + 1) = 2^{n'}(2m' + 1)\]

Suppose that \(n \neq n'\) and assume that \(n < n'\) (without loss of generality). Dividing both sides by \(2^n\) we get:

\[2m + 1 = 2^{n'-n}(2m' + 1) \quad (1.1)\]

Now the lefthand side is odd, but the righthand side is even, so this is not possible, and we have a contradiction. It doesn’t matter whether we take \(n < n'\) or \(n' < n\) we will run into this problem. Thus by contradiction, we have shown that \(n = n'\). If this is so, we divide both sides of Equation 1.4.1 by \(2^n\) and we get

\[2m + 1 = 2m' + 1\]

And thus \(m = m'\). Hence \(\phi\) is injective.

\[\square\]

Before we show that \(\phi\) is also surjective, we observe that if we have a natural number \(k\) of the form \(m2^n\) (with \(m\) being odd), we can recover \(n\) and \(m\) as follows:

\[n := 0; \quad m := k; \quad \text{while even}(m) \land m > 0 \quad \text{do} \quad (m := m/2; \quad n := n + 1) \quad (1.2)\]

This follows our previous discussion about how the result of \(\phi\) can be viewed as a binary number.

**Exercise 1.22**

Why is the program given in 1.2 not a legal \texttt{while} program? Give a corrected version, \textit{i.e.} write an equivalent legal \texttt{while} program.

What this has shown is that there is an inverse function \(\phi^{-1} : \mathbb{N} \to (\mathbb{N} \times \mathbb{N})\). Indeed with a few minor changes you will have written a program to calculate this function in Exercise 1.22.
1.4. CODING DATA STRUCTURES

Lemma 1.4.3

\( \phi \) is a surjection i.e., \( \forall (j : \mathbb{N}) : \exists (n, m : \mathbb{N}) : \phi(n, m) = y. \)

Proof of 1.4.3

Given a natural number \( j \in \mathbb{N} \), we run the program of Exercise 1.22, with the variable \( k \) set to \( j + 1 \) in the initial state. When this program terminates\(^3\), we will have determined \( n \) and \( m' \) with \( j + 1 = 2^n \times m' \) and \( m' \) is odd. Letting \( m = (m' - 1)/2 \), we have found \( n \) and \( m \) such that

\[ \phi(n, m) = j \]

for any \( j \in \mathbb{N} \).

Now we return to the theorem we were proving: Theorem 1.4.1.

Proof of Theorem 1.4.1

By Lemma 1.4.2, \( \phi \) is injective; by Lemma 1.4.3, \( \phi \) is surjective. Thus \( \phi \) is bijective.

Let’s just recap this, shall we? We have shown how to define a bijection between a pair of natural numbers and the natural numbers.

Example 1.4.4

What is the result of \( \phi(3, 15) \)?

\[
\phi(3, 15) = 2^3(2 \times 15 + 1) - 1
\]
\[
= 8 \times 31 - 1
\]
\[
= 247
\]

\(^3\)We really should show that this program terminates. It does because we are reducing \( m \) at each step, see Section 2.5.
Example 1.4.5

What is the pair represented (or coded) by the number 127? This is asking ‘what values of \( n \) and \( m \) make \( \phi(n, m) = 127? \)’

\[
127 = 2^n(2 \times m + 1) - 1 \\
\Leftrightarrow 128 = 2^n(2 \times m + 1)
\]

Thus \( n = 7 \) and \( m = 0 \) (because \( 2^7 = 128 \)).

The fact that we are able to find the inverse also tells us that \( \phi \) is bijective. Indeed, in general an alternative proof technique to demonstrate that a function \( f \) is a bijection, is to show that there is an inverse function.

Exercise 1.23

Define a bijective function between a triple of natural numbers and the natural numbers. Prove that it is bijective.

1.4.2 Lists

We now briefly consider how to encode lists as natural numbers. Almost all structures of interest can be decomposed into pairs and lists, so once we are done we will have tools that should allow us to encode almost any structure we would like to.

As a recap, a list is a possibly empty finite-sequence of values. We can define lists recursively using the grammar

\[
\text{list} = [] \mid n :: \text{list}
\]

where \( n \in \mathbb{N} \), \([\,] \) is the empty list and :: is the cons operator appending a value to the front of a list.

We can then define a coding of lists \( \varphi \) recursively as follows

\[
\varphi([]) = 0 \\
\varphi(n :: l) = 2^n(2\varphi(l) + 1)
\]

i.e. we encode a list as a pair of the head of the list and the rest of the list. However, notice that we do not use the previous pairing function as this possibly gives the value 0 and we reserve 0 for the empty list. Instead we omit the final \(-1\) to shift the coding up by 1.
Exercise 1.24

Prove that $\varphi$ is bijective by giving an inverse function.

(*) Exercise 1.25

Use your favourite programming language to write the coding functions for pairs and lists. Try and write it generally so it allows you to encode pairs or pairs, lists of pairs, lists of pairs of lists of pairs etc.

1.4.3 Operations on Structures as Arithmetic

As we are able to encode pairs and lists as natural numbers we can argue that we do not need these structures in our language to be able to represent programs that use these structures. This is why we only consider numbers in our language.

Exercise 1.26

Provide a coding function that codes a binary tree as a natural number. Recall that a binary tree can be defined recursively as

$$ btree = empty \ | \ (n, btreen, btreen) $$

You might find it useful to use the coding function from Exercise 1.23. Briefly argue that your function is a bijection.

The rest of the argument that we don’t need these structures is that we can encode any operations on such structures as arithmetic. The two main functions on pairs are first and second, retrieving the first and second values respectively. The two main functions on lists are head and tail returning the first element of a list and the rest of the list respectively. It should not be difficult to see that the above coding functions, and their inverses, give us arithmetic functions for computing these functions on numbers encoding the structures.

1.5 Adding Arrays

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that the contents of this section is not examinable (see explanation below).</td>
</tr>
</tbody>
</table>
As you’ll have noticed, our simple programming language does not have arrays; this is because we do not need them in this course. We can use natural number encodings instead.

However, I have chosen to include an extension of \texttt{while} with arrays, which we will call \texttt{while}_{\text{Arr}}. Note that this language is not examinable i.e. you will not be asked to recall or apply its semantics or any information relating to it from other Chapters. So why include it? In practice, most programming languages do have arrays and when we consider the topic of complexity all of the interesting discussions involve programs including arrays.

### 1.5.1 Extending the Syntax

The extended syntax for \texttt{while}_{\text{Arr}} is as follows:

\[
S ::= \text{v := a | skip | } S_1; S_2 | \text{if } b \text{ then } S_1 \text{ else } S_2 | \text{while } b \text{ do } S \\
b ::= \text{true | false | } a_1 = a_2 | a_1 \leq a_2 | \neg b | b_1 \land b_2 \\
a ::= \text{v | n | } a_1 + a_2 | a_1 - a_2 | a_1 \times a_2 \\
v ::= x | x[a_1, \ldots, a_n]
\]

This adds the notion of \textit{array subscripting}, allowing us to write expressions such as \texttt{a[i], b[i, j], or c[i + 1]} wherever a variable can currently occur in a legal \texttt{while} program.

You may immediately notice that this allows us to write programs that might not make sense, for example

\[
x[x[x]] = x
\]

as an extreme case. To deal with this issue (which we didn’t have previously) we can introduce the notion of \textit{types} (see below). We will assume that we use things in their proper place i.e. use variables for arrays only as arrays.

You may also notice that we do not have any way of checking the size of an array to perform bounds-checking. Here we will assume that arrays are \textit{dynamic}, i.e. they expand automatically, although we also assume that indices are positive. In essence this means that arrays are functions on the natural numbers. To facilitate this we will assume that arrays are initialised to zero. These assumptions are captured by the following semantics.
1.5. ADDING ARRAYS

1.5.2 Extending the Semantics

The first thing we need to do is extend the notion of state so that it can store information about the contents of arrays. To support arrays of arbitrary dimensions we now (i) allow state objects to point to state objects, and (ii) allow states to be indexed by natural numbers. Therefore, if the state should contain an array $x$ with values 5 and 6 at indices 1 and 3 then the state would be

$$[x \mapsto [1 \mapsto 5, 3 \mapsto 5]]$$

and if we had a two-dimensional array $y$ with values 5 and 6 at $[1][2]$ and $[2][1]$ then the state would be

$$[y \mapsto [1 \mapsto [2 \mapsto 5], 2 \mapsto [1 \mapsto 6]]]$$

This is clearly difficult to read but we don't anticipate writing these down very much. Once we have done this the we can update how we evaluate arithmetic expressions with the following rule:

$$\mathcal{A}[x[a_1, \ldots, a_n]] s = \begin{cases} \text{s(x)(A[a_1]) \ldots (A[a_n]) if it is defined} \\ 0 \text{ otherwise} \end{cases}$$

We say that a value $s(x)(A[a_1]) \ldots (A[a_n])$ is defined if

$$\mathcal{A}[a_1] \in \text{dom s(x)} \land \ldots \land \mathcal{A}[a_n] \in \text{dom s(x)}(A[a_1]) \ldots (A[a_{n-1}])$$

i.e. all the links required are there.

We are also required to update the semantics of assignment to include array assignment. We give the case for two-dimensional arrays only; the extension to one or n-dimension arrays is straightforward. We can do this as follows, where we skip some of the details. In essence it is necessary to extract and update all the relevant sub-states, storing the value of $a$ in the innermost state. Here we abuse notation and assume that retrieving a non-existent state returns the empty state.

$$\text{[ass] } < x[a_1, a_2] := a_3, s > \Rightarrow s[x \mapsto s(x)[A[a_1]] \mapsto s(x)(A[a_1][A[a_2] \mapsto A[a_3]])] >$$

1.5.3 A Note on Types

The operational semantics is a dynamic semantics, telling us how the program should execute. Sometimes we also require a static semantics restricting the form of valid programs further than the syntactic rules. This introduces the notion of type and rules to tell us which programs are properly
typed (the valid ones). Introducing such rules is outside the scope of the course but there are books given in the reading list that cover this topic in depth. The important thing to note is that conformance with the syntactic rules is no longer sufficient to check whether a program is a statically valid while program.

If you want to explore the idea of types a little further then I suggest you go and have a look at Russell’s Paradox and see how the notion of types was used to avoid it.

1.5.4 Writing some Programs

For these programs we introduce an additional syntactic shortcut (which we could also have used earlier). Here we define a for construct in terms of while.

\[
\text{\texttt{for } x = n \text{ to } m \text{ do } P} = [\text{\texttt{x := n; while}(x \leq m) \text{ do } (P; x := x + 1)]}
\]

We will use this new construct and the array extension to define some programs. As before, we do this by first presenting them as exercises for you to do and then giving the solutions. In all of the below examples and exercises we assume that arrays are non-empty i.e. their length is > 0. As an additional exercise you could consider what needs to be changed for some programs to make them handle the case where \( n = 0 \).

Example 1.5.1

Write a program in while\textsubscript{Arr} that finds the maximal element in an array. Assume the array is given in variable \( a \) and its length is given in variable \( n \) and places the result in variable \( \text{max} \).

Example 1.5.2

Write a program in while\textsubscript{Arr} that performs binary search on a sorted array to find the index of a given value. Assume the array is given in variable \( a \), its length is given in variable \( n \), and the value being searched for is given in variable \( x \). Use \( r \) to indicate whether the search was successful or not and if successful use \( z \) to output the index of the value in \( x \).
1.5. ADDING ARRAYS

Example 1.5.3

Write a program in \texttt{while} that sorts an array in variable \(a\) (with its length in variable \(n\)) from smallest to largest. The sorting is meant to be destructive and at the end of the program \(a\) should be sorted.

Example 1.5.4

Write a program in \texttt{while} that computes the dot product of two arrays \(a\) and \(b\) and stores it in variable \(dot\), both arrays are assumed to have the same length, given in variable \(n\).

Answer to Example 1.5.1

This program simply performs a linear search of the values in \(a\).

\[
\begin{align*}
\text{max} &= a[0]; \\
\text{for } i &= 1 \text{ to } n \text{ do if } \text{max}>a[i] \text{ then max:=a[i]} \\
\end{align*}
\]

We initialise \(\text{max}\) with the first value of \(a\), using the above assumption that \(n > 0\).

\[
\square
\]

Answer to Example 1.5.2

The idea behind this program is that we repeatedly split the array into one half possibly containing \(x\) and one half definitely not containing \(x\). The variable \(\text{mid}\) will store our current guess for the location of \(x\) and \(lo\), \(high\) define the part of the array with think that \(x\) is in.

In this program I cheat and assume that our language has a division operator, but the program could be rewritten without it.

\[
\begin{align*}
lo &= 0; \\
hi &= n; \\
\text{mid} &= n/2; \\
r &= 0; \\
\text{while } (a[\text{mid}] = x) \land (lo > hi) \text{ do} \\
&\quad \text{if } a[\text{mid}] \leq x \text{ then } lo := \text{mid} \\
&\quad \text{else } hi := \text{mid}; \text{mid} := (hi-lo)/2; \\
&\quad \text{if } a[\text{mid}] = x \text{ then } z := a[\text{mid}]; r:=1
\end{align*}
\]
What is the very important assumption we are making about $a$? As an exercise you could rewrite this program to remove the division operator by using the idea from the previous program to compute division.

Answer to Example 1.5.3

I have chosen to implement bubble sort as it is one of the shortest to write. If you are not familiar with this algorithm then the idea is to pass through the array $n - 2$ times and each time compare each pair of values and swap them if they are in the wrong order.

```plaintext
if n > 1 then
    for i = 0 to n - 2 do
        for j = 0 to n - 2 do
            if a[j] > a[j+1] then
                t = a[j]
                a[j] = a[j+1]
                a[j+1] = t
```

Later (in Chapter 3) we will spend some time discussing the efficiency (complexity) of various sorting algorithms. If the array is already sorted this program will still make the same number of iterations over the array. As an additional exercise you could consider how to update the program so that it stops as soon as it knows the array is sorted.

Answer to Example 1.5.4

The program for this is nice and simple

```plaintext
dot := 0; for i = 1 to n do dot := dot + a[i] * b[i]
```

Exercise 1.27

Extend the solution in Example 1.5.1 to additionally place the minimum element in $z$. 
Exercise 1.28

The solution in Example 1.5.3 provides one possible sorting algorithm but there are many different sorting algorithms. Write a \texttt{whileArr} program for a different sorting algorithm.

Exercise 1.29

Write a program in \texttt{whileArr} that takes two arrays \(a\) and \(b\) (with length in variable \(n\)) and computes the cross product of the two arrays and stores the result in an array in variable \(c\).

Exercise 1.30

Write a program in \texttt{whileArr} that takes two sorted arrays \(a\) and \(b\) with lengths in variable \(n\) and \(m\) respectively and merges them into an array \(c\) such that \(c\) is of length \(n + m\) and \(c\) is also sorted.

Exercise 1.31

Write a program in \texttt{whileArr} that takes two arrays \(a\) and \(b\) (with length in variable \(n\)) and treats them as vectors in an \(n\)-dimensional space and then computes the Euclidian distance between them, storing this in \(z\).

1.5.5 Removing Arrays

As with pairs and lists we can code arrays as natural numbers. We do not give the details here as they are tedious and not important. We just need to be happy that we can do it. The trick is to represent an array as a list of pairs \((i, a)\) where \(i\) is an index into the array and \(a\) is the value at that index. There can only be a finite number of places in an array where its value is non-zero, so the resulting list is finite. Once we have this representation we can use the previous coding functions for pairs and lists. Furthermore, we can extend the coding arbitrarily to nested arrays, arrays of pairs etc.
Chapter 2

Proving while Programs Correct

In this chapter we will show that it is possible to prove that a while program is correct. To do this we must specify what the program is supposed to do. The technique we will use involves specifying the state before and after executing a program. We will explore what this means by way of examples.

It is worth briefly considering what this means and why it is important. What this means is that we do not just test a program over a sample of possible inputs and outputs, put instead check mathematically that it will always deliver the right answers for all possible inputs. We will expand this point later.

The typical use-cases justifying the use of program correctness are safety critical systems and security systems. If you take a look at some of the following websites:

http://www.cse.psu.edu/~gxt29/bug/softwarebug.html
https://www5.in.tum.de/~huckle/bugse.html

you will see numerous examples of software bugs that led to the loss of millions of dollars and even the loss of life. Many of these bugs were due to seemingly simple mistakes i.e. out by one errors\(^1\). Arguably these mistakes could not have been made if the techniques introduced in this chapter had been used.

\(^1\)Some are clearly more complex, for example race conditions in concurrent software cannot be dealt with using techniques in this chapter, but there are formal methods that can be applied to detect such bugs.
Learning Outcomes

At the end of this Chapter you will:

- Be able to write simple specifications in the form of pre and post conditions
- Be able to describe the meaning and role of loop invariants and loop variants
- Be able to describe the difference between partial and total correctness
- Be able to apply axiomatic rules to establish the partial correctness of while programs
- Be able to apply axiomatic rules to establish the total correctness of while programs

2.1 What does Correctness Mean?

We have alluded to this both above and in the introduction. It does not make sense to talk about correctness without knowing what the intended outcome was. There are a number of things we could mean by correctness.

**Syntactic Wellformedness.** It is clear that if while x is not a valid program in while as it is not accepted by the grammar of the language. In this sense it is incorrect. Other kinds of syntactic errors you have probably met are missing semicolons or misspelled keywords. These errors are typically easy to detect as one just needs to check the program against the grammar.

**Properly Typed.** Another (usually) static property of a program is whether it type-checks. For example, in java we cannot write String x = 5; as 5 is not of type String. Type-checking is usually designed to be easy to check, although some languages have more complex type systems. Note that while does not have types as they are not needed to discuss computation.

**Well-Behaved With Respect to Language Properties.** A key example of deviation from this kind of correctness is an ArrayOutOfBoundsException
2.1. WHAT DOES CORRECTNESS MEAN?

in java. The java language tells us that we should not access arrays outside of their defined bounds and doing so is an error. Another example is NullPointerException. In C and similar languages we have a host of more complicated errors involving memory-safety. There are many tools that can be used to check these properties.

**Semantic Correctness.** The program $x:=5$ is syntactically correct, assumedly properly typed and doesn’t seem to break any sensible language rules. But if I meant to type $x:=6$ then the program is not correct. This is the first time we meet an explicit notion of intended outcome. In the above cases the intended outcome was implicit in the programming language. Now we need to make it explicit by writing it down. This is the kind of correctness we return to shortly.

**Termination.** Sometimes we want our programs to finish running. Sometimes we don’t. A program to calculate the 1,000,000th prime number is supposed to terminate, this might be after a long time, but it is supposed to give an answer. A web-server is supposed to run continuously and react to requests. So the idea of whether a program should or should not terminate is also part of a specification. We don’t cover the idea in this course, but often for programs that don’t terminate we want to establish other properties, such as progress and fairness.

In all of the above cases there is a clear notion of what the intention was, whether it was implicit or explicit. Here is a quick reminder.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>A well-formed program is not inherently correct or incorrect, it is only correct or incorrect with respect to a given specification. A program $P$ can be correct for a specification $A$ but incorrect for a specification $B$.</td>
</tr>
</tbody>
</table>

So how do we capture a reasonable notion of semantic correctness for while programs? Recall that programs are functions on states i.e. there is a start state and an end state. The actual behaviours of a program $P$ can be collected as the set

$$\text{Actual}(P) \equiv \{(s_{\text{start}}, s_{\text{end}}) \mid (P, s_{\text{start}}) \Rightarrow s_{\text{end}}\}$$
CHAPTER 2. PROVING WHILE PROGRAMS CORRECT

The intended or desired behaviours can be captured similarly by using two predicates: \( \text{Pre} \) for a precondition on the start state and \( \text{Post} \) for a postcondition on the end state. The intended meaning is that if the start state satisfies \( \text{Pre} \) then the end state should satisfy \( \text{Post} \). The assumption is we only care about what happens to programs starting in states satisfying the precondition. The set of behaviours for this specification is then

\[
\text{Specification}(\text{Pre}, \text{Post}) \equiv \{(s_{\text{start}}, s_{\text{end}}) \mid \text{Pre}(s_{\text{start}}) \rightarrow \text{Post}(s_{\text{end}})\}.
\]

The problem of checking the correctness of \( P \) with respect to \( \text{Pre} \) and \( \text{Post} \) is then (theoretically) the problem of checking whether

\[
\text{Specification}(\text{Pre}, \text{Post}) \subseteq \text{Actual}(P).
\]

However, both of the above sets are normally infinite (at least in theory) so checking this inclusion by testing is not a reasonable approach. This chapter introduces a set of rules that allow us to carry out the check.

### Important

The intention here is to provide an intuition for what comes later. In exercises you will be expected to use the presented axiomatic system rather than make these kinds of informal arguments.

This is not the full picture as we have only considered terminating runs of a program i.e. those that have an end state. Non-terminating runs will not be captured in either of the above sets. This is such an important point that we separate the two cases into two kinds of correctness. If we do not consider non-terminating runs then this is partial correctness (partial because we only consider some of the runs). Otherwise, it is necessary to ensure that all runs terminate, this is called total correctness. Often we want to establish total correctness but sometimes this is not possible (see Section 4.4) or desired (recall the web-server example).

### Important

The problem of checking whether a program satisfies its specification if it terminates is called partial correctness. The problem of checking whether a program satisfies its specification and it always terminates is called total correctness.
2.2 Proving the Division Program is Correct

In this section we will show that the division program from Example 1.2.1 (page 9) is correct. As a reminder this program can be written as follows:

\[
\begin{align*}
  r &:= x; \\
  d &:= 0; \\
  \text{while } y \leq r & \text{ do } (d := d+1; \ r := r-y)
\end{align*}
\]

Important
Note that the purpose of this section is to give you an intuition of how we aim to prove these things. The next sections describe how I actually want you to do it i.e. the approach you should follow in exercises.

2.2.1 Writing the Specification

Before we can continue with the proof we need to establish the specification. You may recall that in this exercise we were only concerned with the case where \( x, y \in \mathbb{N} \land y \neq 0 \). To generalise to the integers we can state this as \( x \geq 0 \land y > 0 \). This is our precondition. Our postcondition is that the result of the division is in \( d \) and the remainder in \( r \). We could write this as \( (y \times d) + r = x \), but this is not specific enough as this could just set \( d = 0 \land r = x \) and be true. Furthermore, \( r \) should be positive, otherwise we could add 1 to \( d \) and subtract \( y \) from \( r \) and continue to get further valid solutions. Therefore, our postcondition is \( x = (y \times d) + r \land 0 \leq r < y \).

Exercise 2.1

Write pre and post conditions for the extended division program given in Exercise 1.6 (page 12).

Exercise 2.2

Write pre and post conditions for the logarithm base 2 program given in Exercise 1.11 (page 13).
2.2.2 Proving Partial Correctness

The first thing we are going to tackle is the while-loop and then build the rest of the proof around this. This is a generally good strategy as proving the correctness of while-loops is generally the difficult bit. Dealing with assignments and conditionals is relatively straightforward as it is clear how these update the state. The problem with while-loops is that we cannot tell statically how many times they will execute, so there is no concrete function between states before and states after the loop.

To deal with this issue we will find an expression that is always true of states before and after the while-loop. If we can do this then we can replace the while-loop by this expression and everything becomes nice again. Such an expression is called a loop-invariant.

**Definition 2.2.1**

A loop-invariant for the statement

```
while b do S
```

is a predicate $P$ which is true of the state at the start and end of each execution of a while-loop body $S$.

It is important to note that a loop-invariant does not mean that the values of variables do not change. It is assumed that they do change, but the expression should be true for every combination of values that they can take whilst the loop is executed. Hopefully this will become clearer soon.

There are many possible loop-invariants for our program. For example,

- $d \geq 0$
- $r \leq x$
- $z = 0$

But to be useful it must involve all of the variables that change. Remember that implicitly we want to replace the while-loop by this expression and if some variables are not mentioned then their effect on the rest of the program will not be captured.

<table>
<thead>
<tr>
<th>Confused?</th>
</tr>
</thead>
<tbody>
<tr>
<td>In case what I am saying causes confusion. We will not actually replace the while-loop by a loop invariant. It is just a way of thinking about the role of loop invariants.</td>
</tr>
</tbody>
</table>
In the case of the division program, we see that variable $d$ increases by 1 every time the loop is executed, whilst variable $r$ has $y$ subtracted each time. The value of $y$ does not change. Indeed, only $d$ and $r$ are changed by the loop. One clue that we need to multiply $d$ by $y$ is that one variable increases by 1 and the other decreases by $y$. So, if we add $r$ to $d \times y$ we have a value that should be constant, let’s call it $K$:

$$K = d \times y + r$$

How can we check that it is constant? Let’s look at what $K$ will be on the next iteration of the loop i.e. when $d$ is 1 larger and $r$ is $y$ smaller:

$$(d + 1) \times y + (r - y) = d \times y + y + r - y = d \times y + r = K$$

So what is the value of $K$? We know that initially $r$ is set to the value of variable $x$ and $d$ is 0. Substituting these into the above gives us

$$K = d \times y + r = 0 \times y + x = x$$

So a possible loop invariant is

$$x = d \times y + r$$

which should not be surprising. However, to be strong enough to establish our postcondition we also need to know that $r$ remains non-negative throughout the loop, so that it is non-negative at the end. Note that this is dependent on $x$ being non-negative at the beginning, which is part of our precondition. This gives a loop invariant of

$$x = d \times y + r \land r \geq 0$$

which can easily be shown to hold on every iteration of the loop.

The rest of the proof is quite straightforward. It is easy to see that given our precondition and the assignments that the loop invariant will hold at the beginning of the loop (this is by construction). It only remains to consider what happens when the loop terminates.

The while-loop terminates when $\neg(y \leq r)$, i.e. $r < y$. Thus, when the program finishes the following condition is true

$$d \times y + r = x \land 0 \leq r < y$$

and as this is the postcondition we wanted to hold we have established that the program is (partially) correct.
2.2.3 Proving Total Correctness

To complete the proof that our division program is correct, we must show that each while-loop will always terminate. To do this we must find a quantity that strictly decreases every time we execute the loop and is bounded below i.e. eventually stops decreasing.

One simply scheme that ensures this is to provide an expression $v$ which is a natural number quantity (thus $v \geq 0$) and where each time we execute the loop body this quantity always becomes smaller. Thus whatever value $v$ has when we start, we know that the maximum number of times we can go around the loop is $v$ (why?), and thus the loop must terminate. This quantity $v$ is known as the loop variant.

Definition 2.2.2

A loop-variant for the statement

```plaintext
while b do S
```

is a quantity $v$ which strictly decreases on every iteration of the while-loop and is bounded below by $b$ i.e. when it reaches a certain value the while-loop will terminate.

Once more let us consider our division algorithm. There is one obvious quantity that looks suitable and that is $r$. Each time we execute the loop body this quantity appears to be reduced by $y$. In fact $r$ is only being reduced if the quantity $y$ is positive; if $y = 0$, we are calculating $x \div 0$, and the program will not terminate, which is arguably mathematically reasonable, although extremely poor practical programming! The important feature is that the loop will terminate when $r$ gets small enough, so by making sure $r$ is strictly decreasing we know that we will eventually stop looping.

Given our precondition $x \geq 0 \land y > 0$ we can show that the division algorithm is totally correct using a combination of our partial correctness result and the above argument that it is always terminating.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note that without the precondition $y &gt; 0$ the program is not totally correct but is still partially correct as it still computes division correctly whenever it terminates but there are cases where it will not terminate.</td>
</tr>
</tbody>
</table>
2.3 An Axiomatic System for Partial Correctness

We can be more systematic about proving programs are correct, by using a system to ensure we do not miss out important features or forget to prove something important. Do not be misled: in the same way that double-entry book-keeping keeps account records straight despite still needing arithmetic; following an axiomatic schema will not magically eliminate the need for mathematics. Instead, it gives us a structured way in which to present our reasoning about programs.

Pre- and Post-Conditions and Hoare Triples

Recall that we use predicates on states as pre and post conditions. We say that a state $s$ satisfies a condition $P$ if $P(s)$ holds. That is: $P$ is a predicate (or boolean function) on states. We can give $P$ a type: State $\rightarrow \mathbb{B}$. We will specify the behaviour of a program $S$ in terms of the conditions that are true before and after the execution of the program.

Definition 2.3.1

A Hoare Triple for partial correctness is a triple consisting of two predicates, and a statement, which we will write as:

$$\{P\} \ S \ \{Q\}$$

We will refer to $P$ as the pre-condition, and $Q$ as the post-condition for the statement $S$.

The Hoare triple $\{P\} \ S \ \{Q\}$ says that if we start in a state satisfying $P$ and execute $S$ then we will end in a state satisfying $Q$, or we will not terminate (as this is partial correctness).

Example 2.3.2

Here are some examples of Hoare triples that are valid:

- $\{ \ \text{true} \ \} \ x := 0 \ \{ \ x = 0 \ \}$
- $\{ \ x = 0 \ \} \ x := x + 1 \ \{ \ x = 1 \ \}$
• \( \{ P \} \) if \( x > y \) then \( z := x \) else \( z := y \) \( \{ P \land z = y \} \) where
\( P \) is \( (x = 1 \land y = 2) \). This is an example of how we sometimes
introduce names to make things more readable.

• \( \{ x = 0 \land y = 1 \} \) \( t := x; x := y; y := t \) \( \{ x = 1 \land y = 0 \} \)

Here are some examples of Hoare triples that are not:

• \( \{ x = 1 \} x := 0 \{ x = 1 \} \)
• \( \{ x = 1 \} \) skip \( \{ x = 0 \} \)
• \( \{ x = 1 \} \) if \( x > 1 \) then \( x := x - 1 \) \( \{ x = 0 \} \)

To check the validity of such statements we introduce a set of rules that
allow us to break down the program \( S \) and check each statement individually.
We introduce these rules next. But first let us spend a little more time
considering how we write specifications as Hoare triples.

Example 2.3.3 (Specifying Division)

We can now write the pre and postconditions we generated in the
previous section for Example 1.2.1 as the following Hoare triple

\[ \{ x \geq 0 \land y > 0 \} \quad S \quad \{ z = (x \times d) + r \land 0 \leq r < y \} \]

Note that this is a specification and any program \( S \) satisfying it meets
our requirements. The specification is separate from the proof of a
particular program.

Example 2.3.4 (Initial Values)

Imagine we wanted to specify a program that swapped the values of
variables \( x \) and \( y \). How should we specify this? We might be tempted
to write

\[ \{ \} \quad S \quad \{ x = y \land y = x \} \]

but this does not mean what we want. The program \( x := 1; y := 1 \)
satisfies this specification. Notice that we need to allow for the pro-
gram updating the values of \( x \) and \( y \). To do this we introduce auxiliary
variables to record their values at the beginning of the program. The
solution using auxiliary variables is the following Hoare triple

\[ \{ x = a \land y = b \} \quad S \quad \{ x = b \land y = a \} \]
where $a$ and $b$ record the initial values of $x$ and $y$ respectively. This is a general issue with specification and if we are being very strict we should always record the initial values of variables if we want to refer to them again later. This observation might make us revisit our previous specification of division and update it to be more defensive, but we will keep this previous specification as it is easier to work with.

**Exercise 2.3**

Write Hoare triples capturing the pre and post conditions for the problems described in the following previous Examples and Exercises. Make sure that you capture all implicit requirements and think carefully about whether you need to make use of auxiliary variables.

- The square root problem in Example 1.2.2 (page 9)
- The primes problem in Example 1.2.5 (page 10)
- The extended division problem in Exercise 1.6 (page 12)
- The factorial problem in Exercise 1.8 (page 13)
- The power problem in Exercise 1.9 (page 13)
- The logarithm problem in Exercise 1.11 (page 13)

If you introduce any additional mathematical functions (e.g. $\text{max}$) then they should be defined.

**Inference Rules**

We introduce a proof system that allows us to establish the validity of Hoare triples. This is *sound* (i.e. it does not allow us to prove false things valid) but due to the undecidable\(^2\) nature of the problem the system is necessarily *incomplete* (i.e. there are true things we cannot prove are valid).

The inference rules for the proof system are given in Table 2.7. Two of the rules have no premise and the reset are of the form

\[
\begin{array}{c}
\text{premise} \\
\hline
\text{conclusion}
\end{array}
\]

which can be read as *if the premise is true then the conclusion is true*. I believe you met similar inference rules in COMP11120.

\(^2\)For a discussion of what undecidability means see Section 4.3.
CHAPTER 2. PROVING WHILE PROGRAMS CORRECT

\[
\text{ass}_p \quad \{ P[x \mapsto A[a]] \} \quad x := a \quad \{ P \}
\]

\[
\text{skip}_p \quad \{ P \} \quad \text{skip} \quad \{ P \}
\]

\[
\text{comp}_p \quad \frac{\{ P \} \quad S_1 \quad \{ Q \}, \quad \{ Q \} \quad S_2 \quad \{ R \}}{\{ P \} \quad S_1; \quad S_2 \quad \{ R \}}
\]

\[
\text{if}_p \quad \frac{\{ P \land B[b] \} \quad S_1 \quad \{ Q \}, \quad \{ P \land \neg B[b] \} \quad S_2 \quad \{ Q \}}{\{ P \} \quad \text{if } b \text{ then } S_1 \text{ else } S_2 \quad \{ Q \}}
\]

\[
\text{while}_p \quad \frac{\{ P \land B[b] \} \quad S \quad \{ P \}}{\{ P \} \quad \text{while } b \text{ do } S \quad \{ P \land \neg B[b] \}}
\]

\[
\text{cons}_p \quad \frac{\{ P' \} \quad S \quad \{ Q' \}}{\{ P \} \quad S \quad \{ Q \} \quad \text{if } P \Rightarrow P' \text{ and } Q' \Rightarrow Q}
\]

Table 2.7: Inference System for Partial Correctness of while

With the exception of one rule (the rule of consequence) these tell us how to deal with each kind of statement in our language. The rules form Hoare’s Axiomatic Semantics for partial correctness. It is called a semantics as, like our previous operational semantics, it can be used for defining the meaning of programs. In the following we introduce and explain the rules but I leave fully worked examples to the end of the section.

skip

This is the most straightforward rule. If the pre-condition \( P \) is true for the initial state \( s \), and we execute the \text{skip} statement, then \( P \) will also be true for the final state (since it is also \( s \)).

Sequences with ;

Sequences are the glue that hold programs together. I explain this rule next as they are also the glue that hold our proofs together. Sequences are where
the postcondition of one statement becomes the precondition of the next.

If we have rules for $S_1$ and $S_2$, then we can produce a rule for $S_1; S_2,$
by treating the post-condition of $S_1$ as the pre-condition of $S_2$. This can be
written out as

$$
\begin{align*}
\text{if} \quad & \{P\} S_1 \{Q\} \\
\text{and} \quad & \{Q\} S_2 \{R\} \\
\text{then} \quad & \{P\} S_1; S_2 \{R\}
\end{align*}
$$

Notice how this is similar in structure to the operational view of sequencing. This should not be surprising as $P$, $Q$ and $R$ represent sets of states.

**if-then-else**

Suppose that the pre-condition $P$ is true for the initial state $s$, and that we execute the statement

$$
\text{if } b \text{ then } S_1 \text{ else } S_2.
$$

Further suppose that $Q$ is the post-condition. Then there are two cases to consider:

$B[b]s$ is true In this case we execute statement $S_1$, and as pre-condition
we have one more fact: that $B[b]s$ is true. Thus we require:

$$
\{P \land B[b]\} S_1 \{Q\}
$$

$B[b]s$ is false In this case we execute statement $S_2$, and as pre-condition
we have one more fact: that $B[b]s$ is false.

$$
\{P \land \neg B[b]\} S_2 \{Q\}
$$

Puting this all together, we can say that:

$$
\begin{align*}
\text{if} \quad & \{P \land B[b]\} S_1 \{Q\} \\
\text{and} \quad & \{P \land \neg B[b]\} S_2 \{Q\} \\
\text{then} \quad & \{P\} \text{ if } b \text{ then } S_1 \text{ else } S_2 \{Q\}
\end{align*}
$$
while-do

The while-loop construct is not too tricky to understand, but is the only one which cannot be automated.

As we discovered earlier, there is a special name for the pre-condition for this statement: a loop-invariant. This is the key to writing correct imperative code. It is called an invariant because every time we start and finish the body $S$ of the while-loop $\text{while } b \text{ do } S$ this loop-invariant condition is true.

However, we should take account of the relevant boolean conditions at each stage of the execution. If the loop-invariant is an invariant, then at the beginning of each execution of the loop body $S$, we know in addition that the boolean condition $b$ must be true. At the end of the execution of the loop, the invariant must also be true. Thus

$$\{ P \land B[b] \} \ S \ P$$

At the beginning of the execution of $\text{while } b \text{ do } S$, the loop-invariant must be true, and it is also true at the end. In addition, if the loop terminates then the boolean condition must be false.

$$\{ P \} \text{ while } b \text{ do } S \{ P \land \neg B[b] \}$$

Putting it all together:

$$\text{if } \{ P \land B[b] \} \ S \ P \ \text{then } \{ P \} \text{ while } b \text{ do } S \{ P \land \neg B[b] \}$$

Assignment $x := a$

You’ll notice that so far we have not mentioned assignment $x := a$. This might seem strange as it is seemingly one of the more simple constructs. However, its inference rule is somewhat counterintuitive.

Before we continue let us quickly introduce some extra notation. We will write $P[a \mapsto b]$ for the predicate $P$ with all expressions $a$ replaced by the expression $b$.

At first it would seem that we might want to write

$$\{ P \} \ x := a \ \{ P[x \mapsto A[a]] \}$$

i.e. assignment updates the value of $x$ after the assignment. However, let us consider the following Hoare triple that can be shown to be valid using this rule

$$\{ x = 1 \} \ x := 2 \ \{ 2 = 1 \}$$
2.3. AN AXIOMATIC SYSTEM FOR PARTIAL CORRECTNESS

this is clearly wrong. An alternative might be

\[ \{ P \} \ x := a \ \{ P[a \mapsto x] \} \]

i.e. replace \( a \) by \( x \). But this allows us write false things such as

\[ \{ x = 1 \} \ x := 0 \ \{ x = 1 \} \]

due to the fact that \( 0 \) does not appear in \( x = 1 \).

So what should the correct rule be? Let us suppose that we have an assignment \( x := a \) that takes state \( s \) to state \( s' \) and that we have a postcondition \( P \) that is true of \( s' \). The state \( s' \) will have modified \( s \) to set the value of \( x \) to \( A[a] \). The original value of \( x \) in \( s \) has been lost. This suggests a way to find a suitable precondition. We know that

\[ P(s[x \mapsto a]) \]

is true, from the above. The trick is noticing that this is the composition of three functions

\[ P \circ ([x \mapsto a] \circ s) \]

as the predicate \( P \) is a function from states to boolean values. Due to associativity of function composition this can be rewritten as

\[ (P \circ [x \mapsto a]) \circ s \]

which gives us a function that, when applied to \( s \), gives \( \text{true} \). So the precondition is the postcondition \( P \) applied to the state where \( x \) has its new, modified, value.

This gives us the following rule:

\[ \{ P[x \mapsto A[a]] \} \ x := a \ \{ P \}. \]

As this might still seem a little counterintuitive let’s check it on some examples. Firstly, to find a precondition \( P \) for

\[ \{ P \} \ x := 1 \ \{ x = 1 \} \]

we can apply the rule to give

\[ \{ 1 = 1 \} \ x := 1 \ \{ x = 1 \} \equiv \{ \text{true} \} \ x := 1 \ \{ x = 1 \} \]

which is what we want, as \( x = 1 \) after assigning \( 1 \) to \( x \) after starting in any state. Next consider finding a precondition \( P \) for

\[ \{ P \} \ x := x + 1 \ \{ x = 2 \} \]
by applying the rule we get
\[
\{ x + 1 = 2 \} \ x := x + 1 \ \{ x = 2 \} \equiv \{ x = 1 \} \ x := x + 1 \ \{ x = 2 \}
\]
which is clearly what we want. This is encouraging. Finally, let us check that we don’t have false consequences. Consider finding a precondition \( P \) for
\[
\{ P \} \ x := 1 \ \{ x = 2 \}
\]
by applying the rule we get
\[
\{ 1 = 2 \} \ x := 1 \ \{ x = 2 \} \equiv \{ \text{false} \} \ x := 1 \ \{ x = 2 \}
\]
which says that there are no states such that starting in them and assigning 1 to \( x \) takes us to a state where \( x = 2 \).

Notice that in all of the above cases I have applied the assignment rule backwards. This is typically the easiest way to apply this rule, suggesting that our approach to proof construction should in general be backwards.

As an aside, in the original presentation of this approach there was a forward version of the rule written as
\[
\{ P \} \ x := a \ \{ \exists y : x = a[x \mapsto y] \land P[x \mapsto y] \}
\]
but this is rather difficult to use and we don’t consider it here.

**Rule of Consequence**

There is one further rule, which involves logical operators. This is a very important rule but also a rule that is difficult to use because it is not guided by the structure of the program i.e. for the other rules when we see assignment we need the assignment rule, and similarly for other constructs. But this rule can be applied at any stage.

To motivate the need for this rule consider checking whether
\[
\{ x > y \} \ y := y + 1 \ \{ x \geq y \}
\]
holds. We can work backwards from the postcondition and use the assignment rule to show that
\[
\{ x \geq y + 1 \} \ y := y + 1 \ \{ x \geq y \}
\]
holds. But this doesn’t directly establish what we want to show. Somehow we want to conclude
\[
\{ x \geq y + 1 \} \ y := y + 1 \ \{ x \geq y \} \\
\{ x > y \} \ y := y + 1 \ \{ x \geq y \}
\]
i.e. given something we know is true, the thing we want to be true is true. Here this is quite straightforward as we can use basic mathematical reasoning to show that whenever a state satisfies \( x \geq y + 1 \) it necessarily also satisfies \( x > y \), and vice versa, i.e. the two predicates are equivalent (for the integers, not the reals). Therefore, these two preconditions specify the same set of states.

But now consider

\[
\{ x > y \} \ x := x + 1 \ { x > y } ,
\]

here this collapses to checking that

\[
\begin{align*}
\{ x + 1 > y \} & \ x := x + 1 \ { x > y } \\
\{ x > y \} & \ x := x + 1 \ { x > y }
\end{align*}
\]

holds and it is no longer the case that the two preconditions are true for the same states (consider \([x \mapsto 0, y \mapsto 1]\)). However, we can make the observation that we only need \( x + 1 > y \) to hold for all states satisfying \( x > y \) and not the other direction. This is because in the bottom Hoare triple we are only considering states where \( x > y \) is true, so we will never need to check states where this is not true for any subexpressions.

A symmetric argument can be made for postconditions. Here we can argue that we go in the other direction as any state true at the top must also be true at the bottom. This is all captured in the rule of consequence’s use of implication. If this is still confusing then don’t worry. There are plenty of examples in the rest of this section and Section 2.4 goes into more detail on how to construct proofs using this rule.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notice that the implications go in different directions for the pre and post conditions</td>
</tr>
</tbody>
</table>

We will generally need to use the rule of consequence where we are forced to have one side of the Hoare triple in a different form from the one we want. This can happen when dealing with a given pre or post condition and will often be encountered when dealing with loop invariants.


2.3.1 Some Small Example Proofs

The previous section introduce our rules but we haven’t seen many examples of them being used. To help get us started let’s look at some small proofs.

**Max Value.** As a first example consider the following program that finds the maximum of two variables

\[
\text{if } x > y \text{ then } z := x \text{ else } z := y
\]

Firstly, let us check that it does indeed find the maximum of 1 and 2 by checking

\[
\{ x = 1 \land y = 2 \} \text{ if } x > y \text{ then } z := x \text{ else } z := y \{ z = 2 \}
\]

In the following derivation let \( P \triangleq (x = 1 \land y = 2) \), we then present a partial correctness proof as a proof tree:

\[
\begin{align*}
\{ \text{false} \} & \quad z := x \{ z = 2 \} & \quad \{ P \} & \quad z := y \{ z = 2 \} \\
\{ P \land x > y \} & \quad z := x \{ z = 2 \} & \quad \{ P \land \lnot(x > y) \} & \quad z := y \{ z = 2 \} \\
\{ P \} & \quad \text{if } x > y \text{ then } z := x \text{ else } z := y \{ z = 2 \}
\end{align*}
\]

The leaves of each branch use the assignment rule, the next step down uses the rule of consequence and the first step applies the rule for if then else. Check that you are happy that \((P \land x > y) \rightarrow \text{false}\). Notice that whenever we have a false precondition, e.g. \(\{ \text{false} \} S \{ Q \}\), the Hoare triple will always hold as there will be no resultant states and we can say anything we want about them.

**Important**

Remember that \(\{ \text{false} \} S \{ Q \}\) is always true and that \(\{ P \} S \{ \text{false} \}\) is always false for terminating programs, but true for non-terminating programs.

Now let us prove the program correct with respect to the intended specification, which can be written as following assuming that we have a function
max, which is a reasonable assumption to make.

\[
\{ \quad \text{if } x > y \text{ then } z := x \text{ else } z := y \quad \}\{ z = \max(x, y) \}
\]

In the following derivation let \( M \triangleq z = \max(x, y) \). I am naming certain expressions to make the proofs fit nicely in the page width in tree form. Later we will see a better way of writing proofs that handles larger proofs and has the space to make the rules used more explicit. The proof tree is then

\[
\begin{align*}
\{ x = \max(x, y) \} & \quad z := x \quad \{ M \} \\
\{ x > y \} & \quad z := x \quad \{ M \} \\
\{ y = \max(x, y) \} & \quad z := y \quad \{ M \} \\
\{ \neg(x > y) \} & \quad z := y \quad \{ M \} \\
\{ \quad \text{if } x > y \text{ then } z := x \text{ else } z := y \quad \} & \quad \{ M \}
\end{align*}
\]

The structure is the same. The leaves are instances of the assignment rule but this time we had to be a bit clever about how we applied the rule of consequence. We made use of the fact that \( x > y \rightarrow x = \max(x, y) \) to produce something in the form that we needed to apply the assignment rule.

**Swapping Values.** Another very simple program is the one that swaps the values of \( x \) and \( y \). The specification is that \( x \) and \( y \) should have the other values at the end. As we saw earlier, this involves adding some temporary variables into the specification, which we can write as

\[
\{ x = a \land y = b \} \quad t := x; x := y; y := t \quad \{ x = b \land y = a \}
\]

The proof of this Hoare triple can then be given as the following proof, presented in *linear* form\(^3\) where a single application of a proof rule is given on a numbered line and the tree structure is indicated by referring to the rule used and the relevant lines.

1. \( \{ y = b \land x = a \} \quad t := x \quad \{ y = b \land t = a \} \) by \( \text{ass}_p \)
2. \( \{ y = b \land t = a \} \quad x := y \quad \{ x = b \land t = a \} \) by \( \text{ass}_p \)
3. \( \{ x = b \land t = a \} \quad y := t \quad \{ x = b \land y = a \} \) by \( \text{ass}_p \)
4. \( \{ y = b \land x = a \} \quad t := x; x := y \quad \{ x = b \land t = a \} \) by \( \text{comp}_p(1, 2) \)
5. \( \{ x = a \land y = b \} \quad t := x; x := y; y := t \quad \{ x = b \land y = a \} \) by \( \text{comp}_p(4, 3) \)

This proof makes simple use of the assignment rule and the sequencing rule.

\(^3\)You should be familiar with this from Natural Deduction proofs in COMP11120.
A while-loop. To illustrate the while-loop rule and the use of loop invariants let us consider the Hoare triple
\[
\{ x \geq z \land y \geq 0 \land z \geq 0 \} \textbf{while} x > 0 \textbf{ do } x := x - 1; y := y + 1 \{ y \geq z \}
\]
for a program that adds \( y \) to \( x \) using a while-loop. The first thing to do is to work out what our loop invariant should be. We can look at the pre and postconditions to see what kind of thing might be helpful. These are both in terms of inequalities so we should look for an inequality containing both \( x \) and \( y \). It is a fair assumption that we need to do something with \( x + y \) as this value will be constant due to the semantics of the loop. At the end of the loop we want to know that \( y \geq z \) and we know that \( x \) will be zero, so \( x + y \geq z \) seems like a good candidate. We can also check that this holds at the beginning, which it does. However, we cannot use \( x + y \geq z \) by itself to show that \( y \geq z \) as \( x \) needs to be non-negative. So our loop invariant is \( x + y \geq z \land x \geq 0 \). Coming up with loop invariants can be difficult, and we discuss this further later.

In the following linear proof we will use \( P \triangleq (x + y \geq z \land x \geq 0) \) as our loop invariant.

1. \( \{ x + y + 1 - 1 \geq z \land x - 1 \geq 0 \} \ x := x - 1 \{ x + y + 1 \geq z \land x \geq 0 \} \) by \text{ass}p
2. \( (x > 0) \rightarrow (x - 1 \geq 0) \) by arithmetic
3. \( \{ P \land x > 0 \} \ y := y + 1 \{ x + y + 1 \geq z \land x \geq 0 \} \) by \text{cons}p(1, 2)
4. \( \{ x + y + 1 \geq z \land x \geq 0 \} \ y := y + 1 \{ P \} \) by \text{ass}p
5. \( \{ P \land x > 0 \} \ x := x - 1; y := y + 1 \{ P \} \) by \text{comp}p(3, 4)
6. \( \{ P \} \textbf{while} x > 0 \textbf{ do } x := x - 1; y := y + 1 \{ P \land \neg(x > 0) \} \) by \text{while}p(5)
7. \( (x \geq z \land y \geq 0) \rightarrow (x + y \geq z) \) by arithmetic
8. \( (x \geq z \land z \geq 0) \rightarrow (x \geq 0) \) by arithmetic
9. \( (x \geq z \land y \geq 0 \land z \geq 0) \rightarrow P \) by 7, 8
10. \( (P \land x \leq 0) \rightarrow (P \land x = 0) \rightarrow (y \geq z) \) by arithmetic
11. \( \{ x \geq z \land y \geq 0 \land z \geq 0 \} \textbf{while} x > 0 \textbf{ do } x := x - 1; y := y + 1 \{ y \geq z \} \) by \text{cons}p(6, 9, 10)
2.3. AN AXIOMATIC SYSTEM FOR PARTIAL CORRECTNESS

Notice that this makes heavy use of the rule of consequence. As mentioned earlier, this is typical for proofs using the while-rule where the pre and post conditions differ from the loop invariant.

2.3.2 The Division Program Again

Let us now use this system to prove the partial correctness of our division algorithm. We already have the Hoare triple we want to establish. The program can be proved correct as follows, using our previous loop invariant of $P \triangleq (x = d \times y + r \land r \geq 0)$:

1. $\{ x = (d+1) \times y + r - y \land r - y \geq 0 \} \ d := d + 1 \ \{ Q \} \text{ by ass}_p$
2. $x = (d+1) \times y + r - y = (d \times y) + y + r - y = d \times y + r \text{ by arithmetic}$
3. $y \leq r \rightarrow r - y \geq 0 \text{ by arithmetic}$
4. $\{ P \land y \leq r \} \ d := d + 1 \ \{ Q \equiv (x = d \times y + r - y \land r - y \geq 0) \} \text{ by } \text{cons}_p(1, 2, 3)$
5. $\{ x = d \times y + r - y \land r - y \geq 0 \} := r - y \ \{ P \} \text{ by ass}_p$
6. $\{ P \land y \leq r \} \ d := d + 1; r := r - y \ \{ P \} \text{ by comp}_p(4, 5)$
7. $\{ P \} \text{ while } y \leq r \text{ do } S_3 \ { P \land \neg(y \leq r) } \text{ by while}_p(6)$
8. $\{ x = r \land r \geq 0 \} \ d := 0 \ \{ P \} \text{ by ass}_p$
9. $\{ x \geq 0 \} \ r := x \ \{ x = r \land r \geq 0 \} \text{ by ass}_p$
10. $\{ x \geq 0 \} \ r := x; d := d - 1 \ \{ P \} \text{ by comp}_p(8, 9)$
11. $\{ x \geq 0 \land y > 0 \} \ S \ \{ P \land r < y \} \text{ by comp}_p(7, 10) \text{ (and cons}_p)$

Note how in this proof I have introduced names for subparts of the program and for expressions that will be reused. This is used to generally improve the readability of the proof. When doing this make sure you are careful that you use names consistently. It is also okay to skip some obvious steps. For example, in the last step we implicitly rely on $(x \geq 0 \land y > 0) \rightarrow (x \geq 0)$ and the cons$_p$ rule. But again be careful that they are obvious and if in doubt include everything.
2.3.3 A More Complicated Example

Let us now consider a much more complicated program i.e. the GCD program introduced in Example 1.2.4. To write a specification for this function we will use a mathematical function $\text{gcd}$ that computes the greatest common divisor of two numbers characterised by the following axioms (which we will also need to use in the proof):

$A_1$. $\text{gcd}(a, a) = a$ for $a > 0$

$A_2$. $\text{gcd}(a, b) = \text{gcd}(b, a)$ for $a, b > 0$

$A_3$. $b < a \Rightarrow \text{gcd}(a, b) = \text{gcd}(a - b, b)$ for $a, b > 0$

Using this function we can then write the specification of the problem as follows:

$$\{ x > 0 \land y > 0 \} \ C \ \{ z = \text{gcd}(x, y) \}$$

To make things friendlier here we are going to update the program slightly so that it copies the input variables into auxiliary variables and then operates on these. Here is the updated program:

```plaintext
a := x
b := y
while a != b do (
  if (a>b) then
    a := a-b
  else
    b := b-a
)
z:=a
```

We now need to find a loop invariant. This is quite straightforward here as we can use $\text{gcd}(x, y) = \text{gcd}(a, b)$. To convince ourselves that this is a loop invariant we can consider axiom $A_3$ which tells us that $\text{gcd}(a, b) = \text{gcd}(a - b, b)$ given certain restrictions on $a$ and $b$. As before, we need to strengthen the loop invariant slightly for it to work, and here we have to use $a > 0 \land b > 0$ as this helps us apply the above axioms.

To make the proof a bit more readable we introduce a list of definitions:

- $P_{xy} \triangleq (x > 0 \land y > 0)$
- $P_{ab} \triangleq (a > 0 \land b > 0)$
- $I \triangleq \text{gcd}(x, y) = \text{gcd}(a, b)$
2.3. AN AXIOMATIC SYSTEM FOR PARTIAL CORRECTNESS

- \(Q_1 \triangleq (a > 0 \land y > 0 \land \gcd(x, y) = \gcd(a, y))\)
- \(Q_2 \triangleq (a - b > 0 \land b > 0 \land \gcd(x, y) = \gcd(a - b, b))\)
- \(Q_2 \triangleq (a > 0 \land b - a > 0 \land \gcd(x, y) = \gcd(a, b - a))\)

These are then used in the following linear proof of partial correctness:

1. \(\{P_{xy} \land \gcd(x, y) = \gcd(x, y)\} a := x \{Q_1\}\) by \(\text{ass}_p\)
2. \(P_{xy} \rightarrow (P_{xy} \land \gcd(x, y) = \gcd(x, y))\) by logic
3. \(\{P_{xy}\} a := x \{Q_1\}\) by \(\text{cons}_p(1, 2)\)
4. \(\{Q_1\} b := y \{P_{ab} \land I\}\) by \(\text{ass}_p\)
5. \(\{P_{xy}\} a := x; b := y \{P_{ab} \land I\}\) by \(\text{comp}_p(3, 4)\)
6. \(\{Q_2\} a := a - b \{P_{ab} \land I\}\) by \(\text{ass}_p\)
7. \((P_{ab} \land I \land a > b) \rightarrow Q_2\) by \(A_3\)
8. \(\{P_{ab} \land I \land a > b\} a := a - b \{P_{ab} \land I\}\) by \(\text{cons}_p(6, 7)\)
9. \(a > b \rightarrow a \neq b\) by arithmetic
10. \(\{P_{ab} \land I \land a > b \land a \neq b\} a := a - b \{P_{ab} \land I\}\) by \(\text{cons}_p(8, 9)\)
11. \(\{Q_3\} b := b - a \{P_{ab} \land I\}\) by \(\text{ass}_p\)
12. \((P_{ab} \land I \land b > a) \rightarrow Q_3\) by \(A_2, A_3\)
13. \(\{P_{ab} \land I \land b > a\} b := b - a \{P_{ab} \land I\}\) by \(\text{cons}_p(11, 12)\)
14. \(b > a \rightarrow -(a > b) \land a \neq b\) by arithmetic
15. \(\{P_{ab} \land I \land \neg(a > b) \land a \neq b\} b := b - a \{P_{ab} \land I\}\) by \(\text{cons}_p(13, 14)\)
16. \(\{P_{ab} \land I \land a \neq b\}\) if \(a > b\) then \(X_2\) else \(X_3\) \{\(P_{ab} \land I\)\} by \(\text{if}_p(10, 15)\)
17. \(\{P_{ab} \land I\}\) while \(a \neq b\) do \(X_1\{P_{ab} \land I \land a = b\}\) by \(\text{while}_p(16)\)
18. \(a > 0 \land a = b\) \(\rightarrow \gcd(a, b) = a\) by \(A_1\)
19. \((\gcd(x, y) = \gcd(a, b) \land a = b) \rightarrow a = \gcd(x, y)\) by 18
20. \(\{P_{ab} \land I\}\) while \(a \neq b\) do \(X_1\{a = \gcd(x, y)\}\) by \(\text{cons}_p(17, 19)\)
21. \{ a = \text{gcd}(x, y) \} \ z := a \ \{ z = \text{gcd}(x, y) \} \text{ by ass}_p

22. \{ P_{xy} \} \ GCD \ \{ z = \text{gcd}(x, y) \} \text{ by comp}_p(5, 20, 21)

Again I implicitly name subparts of the program. Notice the non-trivial use of the above axioms about \text{gcd}. To construct the proof I started with the loop and then worked backwards from the end of the loop to the beginning, applying the conditional and assignment rules. Notice that the structure of the proof is very similar for each branch of the conditional. This is quite common and I could possibly have structured the proof more nicely to demonstrate this.

2.3.4 Final Thoughts and Exercises

It is worth stressing a point that was made earlier, that post conditions only apply to terminating programs. This important so it gets an extra box.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>The partial correctness property is: ‘If the pre-condition is true of the initial state, and the statement ( S ) terminates, then the post-condition must be true of the final state.</td>
</tr>
<tr>
<td>Note very carefully that if the program does not terminate, the postcondition may be true or false.</td>
</tr>
</tbody>
</table>

Exercise 2.4

Explain why we can use any post-condition with a non-terminating program such as

\[ \text{while true do skip} \]

And the partial correctness property will hold.

Before attempting the following exercises you might find it helpful to read Section 2.4.

Exercise 2.5

Prove partial correctness of

\[ \{ x \neq y \land x = a \land y = b \} \ S \ \{ (a > z \lor b > z) \land (z = a \lor z = b) \} \]
where \( S \) is the following program for finding the minimum of two variables

\[
\text{if } x < y \text{ then } z := x \text{ else } z := y
\]

Why did I introduce the auxiliary variables \( a \) and \( b \)?

Exercise 2.6

Prove the partial correctness of the extended division program you wrote for Exercise 1.6 with respect to the specification you wrote for Exercise 2.3. You will find it useful to refer to the above proof of the unextended division program.

Exercise 2.7

Complete the inference tree for partial correctness for the following program which raises 2 to the power of \( x \) (provided \( x \geq 0 \)).

\[
p := 1; \text{ while } 1 \leq x \text{ do } (p := p \ast 2; x := x - 1)
\]

Exercise 2.8

Prove the partial correctness of the logarithm program you wrote for Exercise 1.11 with respect to the specification you wrote for Exercise 2.3.

As further exercises you should take the other programs introduced in Examples and Exercises in Section 1.2.2 and have a go writing their specifications and proving their partial correctness.

2.4 Some More Hints

Proving partial correctness requires two kinds of creative steps:

1. Generating loop invariants; and

2. Applying the rule of consequence to put things together.

The rest of the steps should be relatively straightforward as they are driven directly by the structure of the program. In this section we discuss these two steps in further detail and give some hints on how to apply them.
2.4.1 The Rule of Consequence Again

Applying the rule of consequence can be tricky so I want to give you some hints and some derived rules that might make things clearer. But first I want to revisit what

\[ \{ P \} S \{ Q \} \]

means. If this Hoare triple holds then for any state \( s \) if \( P(s) \) is true then executing \( S \) on \( s \) will produce some state \( s' \) such that \( Q(s') \) is true. I have highlighted any and some because they highlight which states we care about. The important thing to note is that executing \( S \) on states that satisfy \( P \) will not necessarily result in all states that satisfy \( Q \). Another important point is that if we show that the triple holds for more states than those satisfying \( P \) then we are still okay, as those more states include those that satisfy \( P \).

Figure 2.1 attempts to clarify this intuition. \( P \) and \( Q \) represent the sets of states we care about initially i.e. those that satisfy \( P \) and \( Q \). We can safely replace \( P \) by a larger set \( P' \) due to the above argument that establishing the Hoare triple for a more general precondition is okay. We can safely replace \( Q \) by a smaller set \( Q' \) (where \( Q' \subseteq Q \)) as if we start in a state \( s \) satisfying \( P \) and execute \( S \) on \( s \) and end in a state \( s' \) in \( Q' \) then we also have a state that is in \( Q \).

Often we don’t need to both enlarge the set of states satisfying the precondition and shrink the set of states satisfying the postcondition. This suggests that we can split the rule of consequence into two separate derived rules:

\[
\begin{align*}
\{ P' \} S \{ Q' \} & \quad \text{if } P \rightarrow P' \\
\{ P \} S \{ Q \} & \quad \text{if } P' \rightarrow P \\
\{ P' \} S \{ Q' \} & \quad \text{if } Q' \rightarrow Q \\
\end{align*}
\]
The rule on the left is called *precondition strengthening* and the rule on
the right is called *postcondition weakening*. We use the term strengthening
because the assumed direction of the proof is upwards. In mathematics $B$
is *stronger* than $A$ if $B$ holds whenever $A$ holds i.e. $A \rightarrow B$. So in
the precondition strengthening rule $P'$ is stronger than $P$ and we replace $P$
by $P'$ as we move upwards. A symmetric motivation is used for the name of
the postcondition weakening rule.

So our notion of bigger and smaller is captured by implication. If you
struggle to see why $P \rightarrow P'$ means that $P$ must be included in $P'$
then consider the picture in Figure 2.2. The shaded bits are the things that are
true for $P \rightarrow P'$ (recall that this is equivalent to $\neg P \lor P'$). If $P'$
does not include $P$ then $P \rightarrow P'$ cannot be true, as for it to be true we need to be
able to shade the full box.

There is a common case that deserves a special rule. Consider the proof

\[
\{ x = x \} \ x := 1 \ { x = 1 } \\
\{ \text{true} \} \ x := 1 \ { x = 1 }
\]

where we use the rule of consequence to go from the top application of the
assignment rule to the Hoare triple we want to hold at the bottom. Here it
is not just the case that $P \rightarrow P'$ but $P \leftrightarrow P'$ i.e. $P$ and $P'$ represent the
same set of states. This happens most often when we remove equivalences,
and we can define the following rules for this:

\[
\begin{align*}
\{ P \} S \{ Q \} & \quad \{ P \} S \{ Q \} \\
\{ P \wedge E = E \} S \{ Q \} & \quad \{ P \wedge E = E \} S \{ Q \}
\end{align*}
\]

Here we are allowed to delete equivalences in directions that the rule of consequence does not normally allow, this is because the resulting triples are true for an equivalent set of states.

\textbf{(*) Exercise 2.9}

Prove that the derived rules given here are sound i.e. only allow us to derive things that are true. Hint: consider which states they apply to.

As applications of these rules often require you to find valid implications here are some that might be useful:

- \((a \land b) \to a\)
- \(a \to (a \lor b)\)
- \(a > b \to a \geq b\)
- \(a > b \to a + 1 > b\)
- \(a > b \leftrightarrow a \geq b + 1\)
- \(a \geq b \to (a > b \lor a = b)\)
- \((a + b > 0 \land b \geq 0) \to a > 0\)

You should replace \(a\) and \(b\) by useful things and note that many of these have symmetric versions.
2.4.2 Finding Loop Invariants

I introduced the above derived rules before this short discussion of how to find loop invariants because the most common way to find useful loop invariants is to start with something we already have and either strengthen it or weaken it.

In general, whilst constructing a proof we will reach a point where we have a while-loop and there will be an expression that is currently true before the loop and an expression that is currently true after the loop and we have to find something that connects the two. In other words, we are faced with the challenge of proving

\[
\{ A \} \textbf{while } b \textbf{ do } \{ B \}
\]

using the rule

\[
\begin{array}{c}
\{ P \land \mathcal{B}[b] \} S \{ P \} \\
\{ P \} \textbf{while } b \textbf{ do } S \{ P \land \lnot \mathcal{B}[b] \}
\end{array}
\]

If we find an appropriate loop invariant $P$ we will then need to show that $A \rightarrow P$ and $(P \land \lnot \mathcal{B}[b]) \rightarrow B$ i.e. we will need to weaken $A$ to get $P$ and strengthen $(P \land \lnot \mathcal{B}[b])$ to get $B$. This gives us a strategy for finding $P$ i.e. start with either $A$ or $B$ (we might only have one at this stage of the proof, probably $B$ if we are going backwards) and strengthen/weaken it until things work.

Notice, also that we are going to have to use $P$ to establish

\[
\{ P \land \mathcal{B}[b] \} S \{ P \}
\]

which might suggest that if we are going backwards from $P$ to $P \land \mathcal{B}[b]$ that we need to make $\mathcal{B}[b]$ hold at the start of the loop. However, note that

\[
\begin{array}{c}
\{ P \} S \{ P \} \\
\{ P \land \mathcal{B}[b] \} S \{ P \}
\end{array}
\]

by precondition strengthening as $(P \land \mathcal{B}[b]) \rightarrow P$.

2.5 An Axiomatic System for Total Correctness

If, in addition to being partially correct, we prove that every loop must terminate then we say that the program $S$ is \emph{totally correct} with respect to the pre and postconditions.
We use a different syntax for Hoare triples to indicate that we are talking about total correctness.
Definition 2.5.1

A Hoare Triple for total correctness is a triple consisting of two predicates, and a statement, which we will write as:

\[ [P] S [Q] \]

We will refer to \( P \) as the pre-condition, and \( Q \) as the post-condition for the statement \( S \), just as we did for partial correctness.

The only difference between the axiom system for partial correctness and the new one for total correctness lies in the rule for while-loops. For the other parts of the language we just add rules such as

\[
\{ P \} S_1; S_2 \{ Q \} \\
[ P ] S_1; S_2 [ Q ]
\]

but I won’t write these all out because that would be boring.

Instead, let’s consider the while rule. As we have already seen, the idea is to have an expression (called \( E \) here) that (i) gets smaller after executing the loop, and (ii) will be non-negative whenever we execute the loop. It should be clear that the combination of these two facts implies that there are a bounded number of iterations of the loop e.g. the starting value of the expression \( E \). If this is not clear then note that (ii) ensures that we cannot keep decreasing \( E \) forever. The rule is given as:

\[
[ P \land B[b] \land E = n ] C [ P \land E < n ] \quad \text{if } P \land B[b] \rightarrow E \geq 0 \\
[ P ] \text{while } b \text{ do } C [ P \land \neg B[b] ]
\]

Notice that this just gives us something else to establish as well as the loop invariant. The expression \( E \) what was previously defined as a loop variant.

2.5.1 Division Again

In this course we will not consider complicated total correctness proofs. It is sufficient to see a proof for a relatively simple program that has a simple loop variant. For this we will use the division algorithm again. Remember that earlier we noticed that it has a loop variant \( r \). We can use this to establish

\[
[ x \geq 0 \land y > 0 ] S [ P \land r < y ]
\]

where \( S \) is our division program, which is
Theorem 2.1: Proving While Programs Correct

r := x; d := 0; while y ≤ r do (d := d+1; r := r−y)

in case we had forgotten! Again we define some useful predicates to help with our proof

- \( P \triangleq (x = d \times y + r \land r \geq 0 \land y > 0) \)
- \( Q_1 \triangleq (x = d \times y + r - y \land r - y \geq 0 \land r = n) \)
- \( Q_2 \triangleq (x = d \times y + r - y \land r - y \geq 0 \land r - y = n + y) \)

noting that \( Q_1 \equiv Q_2 \) but we use both to show the effect of the assignment rule at one point. The linear proof can then be given as follows.

1. \([x = (d+1) \times y + r - y \land r - y \geq 0 \land r = n] d := d + 1 [Q_1] \) by ass
2. \( x = (d+1) \times y + r - y = (d \times y) + y + r - y = d \times y + r \) by arithmetic
3. \( y \leq r \rightarrow r - y \geq 0 \) by arithmetic
4. \([P \land y \leq r \land r = n] d := d + 1 [Q_1] \) by const(1, 2, 3)
5. \([Q_2] := r - y [P \land r = n + y] \) by ass
6. \([P \land y \leq r \land r = n] d := d + 1; r := r - y[P \land r = n + y] \) by comp(4, 5)
7. \( (r = n + y \land y > 0) \rightarrow r < n \)
8. \([P \land y \leq r \land r = n] d := d + 1; r := r - y [P \land r < n] \) by const(6, 7)
9. \((P \land y \leq r) \rightarrow y > 0 \)
10. \([P] \) while \( y \leq r \) do \( S_3 [P \land \neg(y \leq r)] \) by while(8, 9)
11. \([x = r \land r \geq 0] d := 0 [P] \) by ass
12. \([x \geq 0 \land y > 0] r := x \) \( [x = r \land r \geq 0] \) by ass
13. \([x \geq 0 \land y > 0] r := x; d := d - 1 [P] \) by comp(11, 12)
14. \([x \geq 0 \land y > 0] S [P \land r < y] \) by comp(10, 13) (and cons)

Notice that the main way in which this differs from the partial correctness proof is that we now need the precondition \( y > 0 \), which wasn’t required before. This should not be surprising, as this is the condition that ensured termination. Notice that instead of using \( r < n \) directly I used the expression \( r = n + y \) that resolves to \( r = n \) when applying the assignment rule and then separately showed that this expression implied \( r < n \).
Exercise 2.10

Prove the total correctness of the extended division program you wrote for Exercise 1.6 with respect to a total version of the specification you wrote for Exercise 2.3.

Exercise 2.11

Prove total correctness for the program given in Exercise 2.7

Exercise 2.12

Prove the total correctness of the logarithm program you wrote for Exercise 1.11 with respect to a total version of the specification you wrote for Exercise 2.3.

2.6 What about Arrays?

I am not going to give you reasoning rules for programs with arrays because they are outside the scope of this course. But I will quickly point out why things are tricky. We cannot just write

\[
\{ P[b[A[a_1]] \rightarrow A[a_2]] \} \ b[a_1] := a_2 \ \{ P \}
\]

i.e. use the assignment rule, as this would allow us to prove

\[
\{ x = y \land b[y] = 0 \} \ b[x] = 1 \ \{ x = y \land b[y] = 0 \}
\]

which is clearly false. The problem is that one variable might alias another. This problem is often called the frame problem as we need to know what is allowed to change outside of our frame of references. To solve this problem we need to add a special rule and some extra axioms that allow us to reason about the equivalence of arrays.

2.7 Practical Correctness

---

4If you are very interested then I believe the third year course on Hoare Logic at Cambridge includes arrays and the notes are available online.
This chapter has introduced some fundamental yet powerful techniques for proving the correctness of sequential imperative programs (i.e. most of the programs we write). At this point I point out that there are other powerful techniques for proving the correctness of non-sequential and non-imperative programs, but these are out of scope for a first-year undergraduate course.

However, these techniques need quite a bit of tool support to be practically useful. Such tools do exist, and if you are interested in playing with one then contact me directly and I’ll send you some links, however, I think it would also be useful to show you some practical tools that you can use to assert similar notions of correctness in programs you are already writing.

The tool that I am referring to is the programming language concept of assertions. Let’s look at a practical $7 billion bug: which caused the loss of Ariane-5 Flight 501.

```java
// Calculates the sum of a (int) + b (int) // and returns the result (int).
int sum(int a, int b) {
    return a+b;
}
```

What could possibly go wrong with such simple code? By adding an assertion we can check.

```java
// Calculates the sum of a (int) + b (int) // and returns the result (int).
int sum(int a, int b) {
    assert (Integer.MAX_VALUE - a >= b);
    return a + b;
}
```

The assertion is checking that there is not an integer overflow. With the integers used in java an overflow is likely to return a negative value. In the particular code used in Ariane-5, the issue was the use of 16-bit unsigned integers in the control program, and the new rocket having significantly more powerful engines.

The real beauty of assertions lies in the fact that by default they are not compiled into your code. In java you can switch them on and force the testing of all assertions by using the compile-time flag -enableassertions. So, to recap:
2.7. PRACTICAL CORRECTNESS

- Assertions are as efficient as comments (in that they are not compiled-in unless you force them to be);

- They are an excellent guide to documenting your code (you are adding in-line unit tests to your code); and

- During development, you can switch them on and discover every case where your assumptions about your own code are wrong!

My recommendation is that you use them to document the permitted input values to functions, the expected range of outputs of functions and for the loop variants and invariants for complicated loops. For example, I’d use an assertion to check that the input to a factorial function was non-negative, and another one to check that the calculated answer was a positive integer.

However, you should be careful. Assertions are turned off by default and if you want to report the bad behaviour in production code you should use exceptions and normal error handling mechanisms.

As well as being dynamically checkable, there exist tools for statically checking Java assertions. Again, if you are interested in these then contact me.
Exercise Classes

As usual, make sure you show your working.

Week 6

- Give two reasons each for why we should study correctness, complexity and computability (6 reasons in total)
  - Exercise 1.5 on page 8
  - Exercise 1.6 on page 12
  - Exercise 1.11 on page 13
  - Exercise 1.15 on page 19

Week 7

- Exercise 1.19 on page 20
- Exercise 1.26 on page 25
- Exercise 2.3 on page 43
- Exercise 2.5 on page 56
- Exercise 2.6 on page 57
Exercise Sheet 8 for examples class in Week 9

1. For each of the following specifications, give a program $S$ that satisfies the specification, or if there is no such $S$ explain why.

   (a) $\{ \} S \{ \ x = y \}$
   (b) $\{ x = a \} S \{ z = 2 \times a \}$ where $a$ is not in $S$
   (c) $\{ x = a \land y = b \} S \{ a < z < b \}$ where $a, b$ is not in $S$
   (d) $\{ \ false \} S \{ \ true \}$
   (e) $\{ \ true \} S \{ \ false \}$
   (f) $[ \ true \] S [ \ false \]$
   (g) $\{ x = a \land y = b \land x > y \} S \{ y > x \}$ but not $[ x = a \land y = b \land x > y ] S [ y > x ]$

2. Give a proof for the following partial correctness problem

   $\{ x = a \land y = b \land x \geq 0 \} \textbf{while } x > 0 \ do \ (x := x - 1; y := y + 2) \ { y = b + (2 \times a) }$

   You will need to find a loop invariant for the loop first

3. Give a proof for the following total correctness problem

   $[ x = a \land y = b \land 0 \leq x < (2 \times y) ] \textbf{while } x > 0 \ do \ (x := x - 1; y := y - 2) \ [ y > 0 ]$

   You will need to find both a loop invariant and a loop variant for the loop first

4. Extend the partial correctness proof of the GCD algorithm given in 2.3.3 to show total correctness e.g.

   $[ x > 0 \land y > 0 ] \ C \ [ z = \text{gcd}(x, y) ]$

   You will need to find a loop variant for the loop

5. Exercise 2.12 i.e. prove total correctness for the logarithm program you wrote in Exercise 1.11 with respect to a total version of the specification you wrote in Exercise 2.3