Chapter 1

Semantics: what and why?

In which we encounter several Questions; and fewer Answers — we make our first Acquaintance with the august Madame Algorithm, the exigeant Lord Syntax, and their exuberant Offspring, the young Master Semantics — all of which Introductions being graciously mediated by our venerable old Friend the Right Reverend Arithmetic.

> The first thing you find out when yer dog learns to talk is that dogs don't got nothing much to say. About anything.
>
> Patrick Ness. *The Knife of Never Letting Go.*

What is 3185×247?

It's an odd question to ask, because the answer isn't really interesting. What *is* interesting is that even though you don't know the answer (unless you're a calculating prodigy), you do know how to find the answer.

Learning by rote is dull but unavoidable. It's not much fun, but there are some things that just have to be learnt that way: the letters of the alphabet (26 of them in the UK); spellings (a few thousand for an average adult); numbers (an *awful* lot of them, but the essential ones are the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9); sums ($1 + 1 = 2$, etc., there are a lot of these as well, but the important ones are the hundred sums involving the numbers 0–9); times-tables (usually 12 tables of 12, making 144 multiplications to be learnt); ... and let's stop there.

The difference between letters and spellings on the one hand, and numbers and sums and times-tables on the other hand, is that spelling isn't systematic, whereas numbers and arithmetic *are* systematic. Just as the digits 0–9 can be put together to form all numbers, the letters of the alphabet can be combined to spell out all the words[1]. Although spelling, in most languages, has a few general principles, you usually have to learn how to spell each word you know on an ad hoc basis. Numbers, though, follow a definite pattern, and it's this pattern that spares schoolchildren from having to learn more than twelve times-tables. In fact, they only need to learn ten times-tables, because as soon as you go beyond 9×9, you can use the set of rules for calculating 'long multiplications'. So rather than having to learn an infinite number of multiplication problems, we can learn a few simple rules that let us solve all of these infinitely many problems. Once you've learnt these rules you can multiply any two numbers, and it becomes a rather boring, mechanical calculation.

It's a *mechanical* calculation because machines can do it. You could use a calculator to find the answer, or you could write a program to do it. If you don't have a calculator or a computer handy, you could put yourself into mechanical mode and fall back on the procedures you learnt at school, and you might take pen to paper and produce something like this:

[1] Of course, this only applies to languages like English that have an alphabet.

$$\begin{array}{r} 3\,185 \\ \times \quad 247 \\ \hline 22\,295 \\ 127\,400 \\ 637\,000 \\ \hline 786\,695 \end{array}$$

This breaks down the problem of multiplying 3185 by a big number (247) into the simpler problems of multiplying it by 200, by 40, and by 7, and then adding the results together. And since multiplying by 200 is the same thing as multiplying by 2 and then by 100, and since multiplying by 100 just involves sticking a couple of 0s at the end, all you're really doing is multiplying 3185 by 2, by 4, and by 7, and then doing a bit of adding. Each of these tasks is further broken down into steps: for example, multiplying 3185 by 7 is reduced to multiplying each of the digits 5, 8, 1, and 3 by 7, taking care to write the final digits of the results in the appropriate column and carry over any 10s to the next column.

This procedure that you learnt at school is an example of an *algorithm*: a series of basic operations carried out in a specific order. In this case, the basic operations are multiplying single-digit numbers, and writing and carrying over and adding digits in columns. The digits that are written, carried over, added and multiplied vary from problem to problem, but it is the same algorithm that you use to find the answer to *any* multiplication problem. A simple algorithm — a few simple rules — allows us to solve an infinite number of problems. Now although we use this algorithm to find the result of multiplying two numbers, the algorithm itself doesn't use numbers: it uses *numerals*, and the same is true for the algorithms we learnt at school for adding, subtracting and dividing — they all use numerals.

A numeral is a symbol for a number. We know that a number can be represented by different words in different languages, for example *mitàswi acite nìswi*, *tan ni saba* and *shi san* represent 'thirteen' in Algonquin, Bambara and Chinese. We also know that a number can be represented by different numerals in different *bases*. For example, thirteen is represented in binary (base 2) by the numeral 1101; in octal (base 8) by the numeral 15; in decimal (base 10) by the numeral 13; and in hexadecimal (base 16) by the numeral D. This last one is not to be confused with the Roman numeral for 500 (also D); a Roman would have written thirteen as XIII. The Roman notation is rather odd, as it doesn't use a base, or *place system*, for its numerals: in the Roman system, the symbol 'V' always means five, whereas in decimal notation, '5' might mean five or fifty or five hundred or five thousand, and so on, depending on where it appears (think about the 5 in 1115, or 1151, or 1511, or 5111, etc).

So a number can be represented by different numerals in different bases, just as it can be represented by different words in different languages. We say a numeral *denotes* a number, and it is this relationship of denotation that lies at the heart of semantics. Incidentally, this raises a rather difficult question: what exactly is a number? There is no definitive answer to this question, and it might even seem a rather pointless question; numbers are just those things we count with: 1, 2, 3, and so on. Surely numbers are just the same thing as numerals? We'll see in Chapter 5 that that's actually quite a reasonable position to take, but at this point we don't need to be very precise about what numbers are. We can start to see what semantics is all about by looking at how numbers behave, and how algorithms can use both numerals and this behaviour.

We started with a fairly difficult sum; here are two much easier sums:

$$23\,456 \times 100$$
$$23\,456 \times 101$$

The first of these two sums is easy because we just add two zeroes to the end of the first numeral — we're working purely symbolically here, manipulating the numeral itself. The second is only slightly more complicated: we need only add two zeroes as before, then add the first number to the result, which you could do in your head if you concentrated.

$$\begin{array}{r} 2\,345\,600 \\ + \quad 23\,456 \\ \hline 2\,369\,056 \end{array}$$

We can represent this schematically by saying that if m is the number we're multiplying by 101, then

$$m \times 101 \;=\; m \times 100 \;+\; m \times 1 \;\;.$$

It's not much harder to calculate $23\,456 \times 102$ or $23\,456 \times 1\,001$ using the same trick.

We slipped in a bit of algebra by introducing the name m for the number we had to multiply; we can get even more general by slipping in another couple of names, and noting that what we're exploiting when we do these easy sums is that

$$m \times (n + p) \;=\; m \times n \;+\; m \times p \;\;. \tag{1.1}$$

In the example above, m is $23\,456$ and n is 100 and p is 1. And in the example we started this chapter with, m is $3\,185$, n is 200, and p is 47 — and then, to calculate $3\,185 \times 47$, we repeat with m being $3\,185$, n being 40 and p being 7. All of which boils down to saying that

$$3\,185 \times 247 \;=\; 3\,185 \times 200 + 3\,185 \times 40 + 3\,185 \times 7 \;\;.$$

In fact, we've pretty much come back to where we started and just described the algorithm for long multiplication! We'd just need to be a bit more precise about splitting a number into its individual digits and about writing the results in columns and carrying, but equation (1.1) tells us exactly what's going on. To be more precise: equation (1.1) says why our long-multiplication algorithm gives us the correct answer.

This book is about the semantics of computer programs. Although we haven't said anything yet about programs, we have introduced the three key concepts that we'll keep coming back to in the remainder of this book: syntax, algorithms, and semantics.

Syntax. Numerals are symbols that denote numbers. Words like 'thirteen' can also denote numbers and words can be either written or spoken (or even drawn). Written, spoken, or drawn, we have a symbol that denotes a thing or an idea. But syntax isn't just about individual symbols, it's about how those symbols can combine with other symbols in certain prescribed ways. For example, the numeral '1' can be combined with the numeral '3' to give the numeral '13'; words are combined with other words to produce sentences with complex meanings — and you can't just put any old words together in random order.

Another useful feature of syntax is that it allows you to state problems. Expressions such as '$200 + 47$', or '3185×247', are written symbols, but are more like questions: their answers would be simpler expressions (i.e., numerals) like 247, or 786 695.

Semantics. Syntax is all about symbols; semantics says what's really going on, what those symbols mean: their denotation. The numeral 13, just like the expression $8 + 7$, denotes the number thirteen. Semantics relates syntax to real-world things (though these 'real-world things', such as numbers, might be quite abstract and slippery things). There is a difference between *how* the two expressions '13' and '$8+7$' denote the number thirteen: the first is a systematic use of syntax (the place system); the second is less systematic. A good semantics for numerals should explain and exploit the former's systematicity.

Algorithms. Syntax allows us to write down problems (like $247 \times 3\,185$ or '$8 + 7$') — algorithms tell us how to solve these problems. Examples of algorithms are the procedures we learnt at school for solving arithmetic problems, or the procedures we express in computer programs for sorting arrays, or calculating the average of a sequence of values, or buying a book on-line, and so on. Algorithms use syntax: they tell us how to do *symbolic* computation. Suppose I have some sheep in one field, and some more sheep in another field, and I want to know how many sheep I have all together (perhaps so I know how many turnips to buy as feed for the winter). I might drive all the sheep from the second field and put them into the first field with their fellows, then point at all my sheep and say: *that's* how many sheep I have. Well, in a very real sense, that's how addition works, but it's not much use in working out how many turnips I need. A better solution would be to count the sheep in each field, and then add the two numbers (numerals!) together. The result will be a numeral that I can use in my turnip-buying symbolic calculation.

Algorithms use syntax: they solve a problem that is stated using symbols, and they chop and stir those symbols until they end up with an answer to the problem; and that answer is itself a symbol. Algorithms also — implicitly — use semantics: they rely upon properties of real-world numbers for their *correctness*. It's semantic properties like (1.1) that guarantee that the result I get from adding the numbers/numerals of sheep in each field is the same as I would get if I put all the sheep in one field and counted the total number. We can summarise all this with the following slogan:

Algorithms allow syntax to behave just like the real world.

This syntactic mimicry of the real world is also apparent in *mathematical models*. A mathematical model is an abstraction of the real world that tell us how things will behave. Here's an example of a mathematical model:

$$F = ma \ .$$

Here, F stands for the force applied to some body (for example, the gravitational force applied to an apple by a planet), m stands for the mass of the body (e.g., the apple), and a stands for the acceleration that the body undergoes as a result of the force applied. What this model says is, roughly: 'if you kick something, it'll move'. Of course, there's a bit more to it than that: it allows you to predict in detail how the real world behaves. If you know the values of any two of the quantities force, mass, and acceleration, you can work out the value of the third. For example, given a particular planet that applies a gravitational force, $F = 0.17$ kilogram-metres per second per second, to a particular apple of mass $m = 0.1$ kilograms, we can work out the acceleration of that apple $a = 1.7$ metres per second per second; or given a spaceship of known mass, and the acceleration we want, we can work out how much force we need from its engines. This predictability, this ability to calculate how things need to be set up in order to achieve a desired goal, is what makes mathematical models so useful.

The semantics of computer programs is a mathematical model of how programs behave — what they do. The benefits we gain from this are just what we would expect from any mathematical model: prediction and the ability to analyse. If, say, we want to know whether a program actually does what it's meant to do, or whether two programs do the same thing, we can look to the model for the answer.

1.1 Interlude: Turing and Computability

Sometimes there are stories that are too good to be left untold, even if those stories take us down side-streets and away from the main avenues that take us most directly to our destination, or if their telling demands a lot of technical details that might leave us feeling as though we're lost and holding our map upside-down. We'll put these digressions inside signposted 'Interludes' such as this one. If you're in a hurry, or if you don't like the look of that dimly lit alleyway with its flickering neon signs, then just carry on along the main thoroughfare, and you won't miss anything of critical importance. If you're not in a hurry, and fancy taking in a splash of local colour, feel free to enjoy the detour.

The first mathematical model of computer programs was developed by the English mathematician Alan Turing several years before the first electronic computers were built, and before there were any programming languages. When Alan Turing published his paper *On computable numbers, with an application to the Entscheidungsproblem* [**?**] in 1936, the word 'computer' meant a *person* who carried out calculations. Turing sought to give a precise mathematical characterisation of what a computer could and could not do; in order to achieve this, he stripped the idea of computation down to its bare bones, and came up with what is now called a *Turing machine*.

Computation takes some *input* and produces some *output*. For example, for a multiplication computation the input might be the two numbers to be multiplied, such as 3185 and 247; the output would then be the number 786 695 (and note that both input and output are syntactical). Moreover, computers, whether human or mechanical, usually produce some intermediate results that are used in coming up with the final result. For example, the intermediate results might include the results of multiplying 2×3185, 4×3185 and 7×3185, perhaps set out in columns as on page 1. Turing presented all of this — input, output, and working

memory — in its bare-bones form as a 'tape' divided into cells, each of which could be blank, or could store a zero or a one (another way of simplifying computation to its bare bones is to use binary numerals with just these two digits).

A Turing machine can be thought of as a box that can shuffle along the tape reading and writing ones and zeroes, following a fixed set of instructions. There are a few basic instructions shared by all Turing machines: move one cell forwards or backwards along the tape; read the symbol at the current cell; and erase the symbol at the current cell, or write a zero or a one. However, each individual Turing machine is 'pre-programmed': it carries out sequences of these basic instructions in a fixed order[2]. One Turing machine might carry out the instructions that do multiplication; another might carry out the instructions for addition; yet another might carry out the instructions that compile Java programs; and so on. This might seem to limit the usefulness of Turing machines as a model of computation. After all, modern computers are so useful because they're programmable — wouldn't Turing machines be more useful if we could load programs into them so that they could do multiplication *and* addition, *and* compile Java programs? But Turing didn't invent a programming language for his machines. He didn't need to: he used numbers instead.

The programs that we run on our computers are stored on those computers. Programs are syntactic things: we type them into our program editors as a sequence of characters, taking care to follow the syntactic rules of the programming language we're using. If this language is a compiled language, we can then use the compiler to generate the binary executable. Both the original source code and the binary executable are stored on the computer as a sequence of bytes. Each byte is, of course, just a sequence of eight bits: whether the bytes are representing the ASCII characters of the source code or the instructions of the binary executable, the program is just a sequence of zeroes and ones. Each Turing machine is pre-programmed to execute one algorithm, and just like a program each Turing machine can be represented as a sequence of zeroes and ones. This sequence is a number, called the *description number* of the Turing machine. This is the syntactic aspect of Turing machines: each machine has a 'description' that can be written down on a tape. To get a fully general model of computation, Turing constructed a *universal Turing machine*. The universal Turing machine starts off by reading the first section of its tape, which contains a sequence of zeros and ones that is the description number of some Turing machine; then it behaves exactly like that machine as it reads and writes on the remainder of the tape. In other words, the universal Turing machine is just like a modern digital computer: it can 'load' and execute any program. In fact, this aspect of the universal Turing machine was one of the things that inspired the creation of programming languages and thence stored-program computers when electronic computers finally came to be built.

Mathematical models thrive on simplicity. They're useful because they strip away unimportant details from a complex world to leave just the important details. Turing stripped unimportant details away, leaving just a box on a tape following a few basic instructions in a programmatic way. Armed with this simplicity, and a lot of ingenuity, Turing was able to say exactly what his boxes could do — and also what they could not do. There are some tasks that no Turing machine can do, and since these machines are models of human or mechanical computers, this suggests there are certain limits to what anything — human or machine — can do by following an algorithm. In other words, there are limits to what algorithms can do.

Here's an example of an algorithm:

1. If x is 0, go to (5); otherwise go to (2).
2. Add 2 to y and let this be the new value of y, then
3. subtract 1 from x and let this be the new value of x, then
4. go to (1)
5. stop

In C or Java, we could write this algorithm as

[2]The 'fixed order' might depend upon the zeroes and ones written on the machine's tape, so the Turing machine can react to the input that it's given.

```
while (x != 0) {
   y = y + 2;
   x = x - 1;
}
```

The algorithm repeatedly adds 2 to y as it decrements x and so it seems clear that the algorithm sets y to $Y + 2X$, where X is the starting value of x and where Y is the starting value of y. But this is only the case if the starting value of x is at least 0; if we start off following this algorithm with x initially set to -1, then we'll just keep on repeatedly adding 2 to y and decrementing x, on and on and on because x will never become equal to 0. It's not that there's anything wrong with this algorithm, it's just that its behaviour depends upon the state in which it's begun: it will terminate if x is at least 0, otherwise it'll keep going without end.

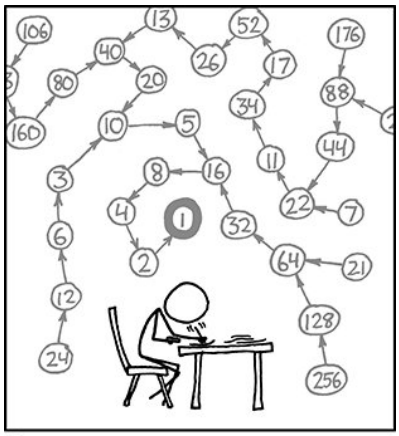
Here's another algorithm:

1. If x is 1, go to (4); otherwise go to (2).
2. If x is even, divide it by 2 and let this be the new value of x; otherwise, multiply it by 3 and add 1 and let this be the new value of x; then
3. go to (1)
4. stop

If, for example, we start off with $x = 7$, then the successive values of x are

$$7 \mapsto 22 \mapsto 11 \mapsto 34 \mapsto 17 \mapsto 52 \mapsto 26 \mapsto 13 \mapsto 40 \mapsto 20 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1 \ .$$

Does this algorithm terminate for any starting value of x? No-one knows. The German mathematician Lothar Collatz conjectured in 1931 that it does, and the Hungarian mathematician Paul Erdös offered $500 as a prize to anyone who could prove Collatz was correct. That money is still up for grabs (but before you reach for your notepad, be warned that Erdös also said that mathematics isn't yet ready for such problems).



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

Figure 1.1: The Collatz Conjecture, `xkcd.com/710`.

These examples suggest a question: is it possible to decide, algorithmically, whether or not a given algorithm will terminate? In other words, could we write a program that takes a program — or more

precisely the description number of a Turing machine — as input, and prints out 'yes' (or '1', say) if the program halts after a finite amount of time, and prints out 'no' (or '0') if the program goes on and on forever? This is called the *halting problem*. Turing showed that the halting problem can't be solved: it's just not possible to write an algorithm to decide whether or not any given algorithm halts. If there were such an algorithm, then we could write *this* algorithm:

1. read in a program and test whether it halts: if it does, go to 4; otherwise, if it doesn't halt, go to 2;
2. go to 3;
3. go to 2;
4. stop.

This algorithm only halts if the program it reads in *doesn't* halt. What happens if we apply this algorithm to *itself*? It reads in itself and tests whether it halts, then it only halts if it itself doesn't halt! This logical paradox all stems from our assumption that we could decide whether or not any given algorithm halts, so clearly this is just not possible.

There are other things that computers can't do; i.e., things for which it's impossible to write an algorithm. These include the following.

- Decide whether or not a given program is a virus, in other words decide whether it makes a copy of itself (for a Turing machine, this would mean that it writes its own description number on its tape).
- Decide whether two programs do the same thing.
- Decide whether or not a given program has a given property (e.g., that it halts, or is equivalent to some other program).
- Decide whether or not a given mathematical statement (e.g., the Collatz conjecture) is true.

This last example is called the *Entscheidungsproblem*[3], and is due to David Hilbert (though it was published in a book co-written with Ackermann [**?**]). Hilbert was one of the great mathematicians of the nineteenth and twentieth centuries, with a great interest in what mathematics is all about, and his contribution isn't just the mathematics he himself did, but also the mathematics he got others to do by posing interesting questions. Two of these questions are:

1. Can mathematics be formalised?
2. Is it possible to decide, following a mechanical procedure, whether a given statement about mathematics is true or false?

The answer to both of these questions is 'no', and in both cases the negative answer relies upon a general property of syntax: formal statements, just like Turing machines, have description numbers. This syntactic property was used by Kurt Gödel (whom we'll meet in later interludes) to explore the limitations of logic, and answer 'no' to the first question, just as Turing used it to explore the limitations of computation and answer 'no' to the second question. If we have a formal definition of what it means to be a virus (such as 'writes its own description number on the tape'), and a formal language in which we can write that definition, then a statement such as 'program p is a virus' also has a description number. If we accept Turing machines as capturing everything that could be considered to be a 'mechanical procedure', then the Entscheidungsproblem boils down to this: is there a Turing machine that starts off with the description number of a mathematical statement on its tape, and writes '1' if that statement is true, and writes '0' if it's false — and can do this for whatever statement we might choose to put on its tape? If there were such a Turing machine, the Collatz conjecture wouldn't be a conjecture (and the Turing machine would be $500

[3]German for 'decision problem'.

to the good), we'd know whether $P = NP$, and mathematicians would only need to think of interesting questions[4].

Anyway, Turing showed that such a Turing machine is an impossibility.

Exercise 1.1 *We haven't been going into much detail, so this exercise is a bit sketchy, but, given that the halting problem is insoluble, can you show that the Entscheidungsproblem is also insoluble?*

Hint: assume that 'program p terminates' can be formalised as a mathematical statement (it actually can be: this was Turing's 'application to the Entscheidungsproblem*').*

1.2 Back to the Introduction

Semantics — mathematical models of programs — are useful for reasoning about what programs do. If you read the interlude above, you'll have seen that such mathematical models are useful for exploring the limits of computation. For example, we know that it's not possible to write a program that decides whether two programs do the same thing, or one that decides whether a program achieves a particular goal. But what else is semantics good for?

Programs are written in order to do something useful: maybe playing a game, calculating safe flight-paths for landing aircraft, or computing tax returns; perhaps rendering video on-screen, steering driverless vehicles, or controlling electrical motors to allow a robot to walk or run. Each program — or potential program — has a specific goal, and stating that goal is the problem of *specification*. Specification usually takes place early on in the software life-cycle, usually before the program itself is written, and is typically followed by a *design* phase, where decisions are made about *how* the program will meet its goal (e.g., what data is required as input, how that data will be represented, etc.), an *implementation* phase, where the program itself is actually written, and a *testing* phase, where — usually — lots of bugs are found. These bugs may point up errors not just in the implementation, but possibly also in the design or specification, and may require changes in some or all of these phases, and possibly a repeat of the whole process. With a little luck, this will lead to a more realistic specification, a better-informed design, a more accurate implementation, and more successful tests; and the software comes closer to its release upon the eagerly waiting world. If the luck runs out, the software-development project might just be abandoned, or released anyway, either not fit-for-purpose or riddled with bugs.

In an ideal world, perhaps, each phase of the software life-cycle would be carried out systematically and with skill and insight, and the ideal world would be full of ideally happy and satisfied purchasers and users of software. In our less-than-ideal world, economics tends to decide what happens. If the incurred costs outweigh the projected gains (or outweigh confidence in meeting the assumptions that led to projecting those gains), the plug will be pulled; if the projected gains exceed the projected costs of releasing buggy code, then buggy code will be released. 'The software crisis' was a phrase coined in the 1970s by Tony Hoare [], to describe the huge amount of software that was released unfit-for-purpose, but, economically, it was really more of a drama than a crisis. Software users very often learn to live with bugs if there's a work-around: if your editor freezes every time you try to save a file to a memory stick, then you might just save your work on the hard drive and use some other software to move it to your memory stick later. In some cases, where human lives depend upon correctly functioning software, the cost of bugs and software failure is unacceptably high. In these safety-critical applications, time (= money) is found in the software life-cycle to give a rigorous justification that the software does what it's meant to do.

What do we mean by 'rigorous justification'? Well, testing certainly isn't good enough. For example, here's a buggy Java method that tests whether a given number `x` is prime:

```
boolean isPrime(int x) {
   return x%2 == 1;
}
```

[4]Which is a large part of their job anyway.

We can test the method by giving it some randomly chosen[5] inputs: a few low numbers, say 3, 4, 5, 6 and 7, then some slightly higher numbers such as 11, 13, 16, 17 and 19, then a spread of larger numbers like 56, 57, 97, 127, 3641 and 674196. Our program passes all of these tests. But that's not a rigorous justification that the program is correct. It can't be, because the program's *not* correct. And that's the problem with testing: failing a test shows — rigorously! — that the program is buggy; passing a test, or a whole suite of tests, doesn't mean a program is free of bugs, it just means we haven't *found* the bugs.

So, suppose we've written the control software for the rockets that will launch, and bring safely back to Earth, the Cute Kittens in Space programme (our sponsors expect the videos of kittens playing with balls of yarn in zero-gee to go viral). Just to have a concrete example, suppose that software involved the following fragment to compute the sum of the numbers $1 + 2 + \cdots + \texttt{n}$.

```
int sum = 0;
int count = 0;
while (count < n) {
   count++
   sum += count;
}
```

We've tested the software, but we're aware of the limitations of testing, and we want to be able to sleep at night without worrying through nightmare scenarios involving cute kittens hurtling through space in a metal box, happily and obliviously patting at balls of yarn just before their craft rebounds off Earth's atmosphere with a calamitous change in velocity.

Here's one way we might ease our febrile night-worries. We note:

1. `count` starts off equal to 0 and is repeatedly incremented until it's equal to `n` (we assume `n` is at least 0);
2. `sum` starts off equal to the sum $1 + 2 + \cdots +$ `count` (the empty sum $1 + 2 + \cdots + 0 = 0$).
3. `sum` *stays* equal to the sum $1 + 2 + \cdots +$ `count`

We'll see in Chapter 6 how we can use semantics to give a rigorous proof that a program is correct Semantics generally comes in three different flavours: operational, denotational, and axiomatic.

Operational Semantics says *how* programs are evaluated

Denotational Semantics gives a mathematical model of programs

Axiomatic Semantics says what is true of programs

1.3 Exercises

Exercise 1.2 *Octal, decimal, and hexadecimal numerals have odd and even digits; for example, 2 is an even digit because any octal, decimal or hexadecimal numeral that ends in 2 represents an even number. In base 3, are there odd and even digits?*

Exercise 1.3 *Write out, as rigorously as you can, the algorithm you learnt at school for adding two numbers together; likewise, write out the algorithm for multiplication.*

Now give these algorithms to a friend and see if they can use this to successfully solve some simple addition and multiplication problems.

[5] Obviously, I'm lying here. This is a variation on the joke about the physicist who thinks that all odd numbers are prime, reasoning that 3 is prime, 5 is prime, 7 is prime, 9 is experimental error, 11 is prime....

Exercise 1.4 *The algorithms that you gave in answer to the previous question depend upon semantic properties of numbers: what, exactly, are those properties?*

Exercise 1.5 *Ancient Egyptians used a different algorithm for multiplication, an algorithm based on doubling. Suppose we want to multiply* 216 *by* 13*. We know that multiplying* 216 *by* 1 *is* 216*; doubling this number tells us that multiplying* 216 *by* 2 *is* 432*; doubling again tells us that multiplying* 216 *by* 4 *is* 864*; and, doubling again, multiplying* 216 *by* 8 *is* 1728*. Now, since* $13 = 8 + 4 + 1$*, we get the answer to our problem by adding* $1728 + 864 + 216$*. This algorithm makes a saving on difficulty by breaking multiplication down to the slightly easier problems of doubling and adding.*

Here's the question: will this algorithm always work for multiplying by 13 or by 17 or by 32 or by 257 or by 975 or by any number whatsoever? And why?

Exercise 1.6 *Here's a program to sum the numbers* $0 + 1 + 2 + \cdots + n$*:*

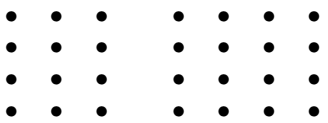
```
int sum = 0;
for (int count = 1; count < n; count++) {
   sum += count;
}
```

What's wrong with this program?

Exercise 1.7 *Here's one to think about: solving the Rubik's Cube involves algorithms. Do those algorithms use syntax?*

Similarly, if you know an algorithm for solving quadratic equations, does that use numbers, or numerals, or what?

Exercise 1.8 *Here's a proof that* $(m + n) \times p = m \times p + n \times p$*:*



How much syntax is used in this proof? — Any semantics? Is it convincing?

1.4 Further Reading

Alan Turing's paper, *On computable numbers, with an application to the Entscheidungsproblem* [**?**], laid the foundations on which Computer Science is built. Read it and marvel at how much can be packed into thirty six pages.

Chapter 2

Syntax

in which is discovered what languages are; and how to define them; also, a means of reasoning about a great variety of things

---

If you know what the word 'syntax' means, you're probably either a programmer or a linguist — or just clever. Programmers learn that word early on in their careers, and it usually comes in painful proximity to the word 'error'. Programming languages set strict rules that define what exactly an acceptable program is, and programmers who don't follow those rules to the letter will have their efforts rejected: they've made a syntax error.

In this chapter, we'll see how to define languages.

2.1 BNF: How to Define a Language

Algol60 report — Jon Backus and Peter Naur: Backus-Naur Form

The Algol committee wanted to define what exactly the acceptable Algol programs are. There are, of course, an infinite number of acceptable programs, and an infinite number of unacceptable programs. What Backus and Naur wanted were rules that would say what was acceptable and what wasn't. Rather than simply stating rules for Algol, they went for the more ambitious goal of inventing a notation that would allow rules to be stated for many different languages.

One thing about languages is that, generally, they're structured, not monolithic: they tend to be made up from different kinds of things. For example, a program in Java is made up of classes, fields, methods, blocks, expressions, variables, and so on; each of these different kinds of things has its own syntax, its own set of rules that say what combinations of characters are acceptable. Methods, for example, must have a return-type, a name (that can't begin with a digit), a (possibly empty) comma-separated list of parameters within round brackets, curly brackets around the body of the method, and so forth. We call these different kinds of things that have their own syntactic rules **syntactic categories**.

Let's look at binary numerals to learn how to specify different syntactic categories in BNF.

2.1.1 Binary Numerals

A binary numeral is a sequence of `0`s and `1`s. These `0`s and `1`s are usually called **digits**, while strings of them grouped together are called **numerals**. These are two different syntactic categories, because there are two different sets of rules that say what acceptable digits are, and what acceptable numerals are. The rules for digits are very simple: `0` is a digit, and `1` is a digit, and nothing else is a digit (not for *binary* digits, anyway).

How would Jon and Peter write these rules down? Here's how:

⟨*BinaryDigit*⟩ ::= `0`
⟨*BinaryDigit*⟩ ::= `1`

Since these are our first examples of BNF rules, let's take a closer look at how they're built up. Every BNF rule has three main components:

- a syntactic category at the start

- the symbol ::=
- and ... some stuff after that.

Let's look at each of these in turn.

Syntactic Categories These are the different kinds of things that have their own syntactic rules. We write these within angle-brackets in a *slanty font*, so that they can be instantly recognised as syntactic categories. In our example, ⟨*BinaryDigit*⟩ is the syntactic category of binary digits. Each rule that begins with a particular syntactic category describes what the acceptable strings of that category are.

The Symbol ::= This can be pronounced as 'can consist of'. Not much more to say about this; it's really just a way of separating the syntactic category at the start of the rule from the stuff that follows.

Some Stuff After That This describes the strings that are acceptable for the syntactic category at the start of the rule. This might consist of actual strings, syntactic categories, or a mixture of the two. In the two rules above, we have actual strings. We write characters using a `teletype font`, such as you might see in a text editor, so that these are recognisable as such. For example, the first rule above says that the string `0` is an acceptable binary digit; the second rule says that the string `1` is also a binary digit.

Before we look at the other options for 'stuff after that', let's review what the two rules we've seen say. The first rule,

⟨*BinaryDigit*⟩ ::= `0`

says that a binary digit can consist of `0`. That is, `0` is an acceptable string for a binary digit. Similarly, the second rule says that `1` is also a binary digit. There is also an implicit addition to any set of BNF rules: the statement 'and that's all!' This means that our two BNF rules have given an exhaustive list of all the ways of writing down acceptable binary digits: we've got `0`, we've got `1`, and nothing else is a binary digit.

Our two rules tell us exactly what binary digits are, but why use two rules when one will do? Jon and Peter also provided an alternative notation that allows us to collapse several rules into one, using the symbol '|' (which we pronounce *or*). The two rules we gave above can be collapsed into the one rule:

⟨*BinaryDigit*⟩ ::= `0` | `1`

which we pronounce 'a binary digit can consist of `0` or `1`' (and don't forget the implicit 'and nothing else').

We call a set of BNF rules a **BNF grammar**; our two rules (or just the one collapsed rule) give us a BNF grammar for binary digits. In fact what we've done is define the *language* of binary digits. Computer scientists think of languages as sets of strings. The language defined by a BNF grammar is the set of acceptable strings we get from its rules. The language of binary digits is the set $\{\texttt{0}, \texttt{1}\}$. To be more precise, a language is generated by a syntactic category; because our set $\{\texttt{0}, \texttt{1}\}$ is generated by the syntactic category of binary digits, we'll refer to this set as $\mathcal{L}\langle \textit{BinaryDigit} \rangle$.

Interlude. Now that we've introduced the notation '$\mathcal{L}\langle \textit{BinaryDigit} \rangle$', we can see that the BNF rule

⟨*BinaryDigit*⟩ ::= `0`

actually says that

$$\texttt{0} \in \mathcal{L}\langle \textit{BinaryDigit} \rangle \tag{2.1}$$

(i.e., the string `0` belongs to $\mathcal{L}\langle \textit{BinaryDigit} \rangle$) and, similarly, the BNF rule

⟨*BinaryDigit*⟩ ::= `1`

actually says that

$$\texttt{1} \in \mathcal{L}\langle \textit{BinaryDigit} \rangle \tag{2.2}$$

Now, there are lots of sets that contain both the string `0` and the string `1` ($\{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{stuff}, \texttt{78}\}$, for example). The implicit 'and nothing else' that we've been going on about says that the language being defined by the BNF rules is the *smallest* set such that (2.1) and (2.2). The smallest set that contains the strings `0` and `1` is obviously $\{\texttt{0}, \texttt{1}\}$, and *that's* why $\mathcal{L}\langle \textit{BinaryDigit} \rangle = \{\texttt{0}, \texttt{1}\}$.

What about binary numerals, then? A numeral is a sequence of one or more digits, so we want some way of saying exactly what such a sequence of digits is. At its simplest, a sequence of one or more digits consists of just one digit on its own (that's a rather short sequence, but it's a sequence nonetheless). So one way of writing down an acceptable numeral is just to write down a digit:

$$\langle \textit{BinaryNumeral} \rangle \ ::= \ \langle \textit{BinaryDigit} \rangle$$

According to our rules of pronunciation, this rule can be read as 'a binary numeral can consist of a binary digit.' Since the binary digits are `0` and `1`, this says that `0` and `1` are also binary numerals, or $\mathcal{L}\langle \textit{BinaryDigit} \rangle \subseteq \mathcal{L}\langle \textit{BinaryNumeral} \rangle$. If this was the only rule we gave for binary numerals, the implicit 'and nothing else' would mean that the languages would actually be equal rather than '$\subseteq$' — so we need another rule: one that tells us that numerals can consist of sequences of more than one digit. There's more than one way of doing this, and one way is to say that a numeral can also be written by writing down a numeral followed by a digit:

$$\langle \textit{BinaryNumeral} \rangle \ ::= \ \langle \textit{BinaryNumeral} \rangle \, \langle \textit{BinaryDigit} \rangle$$

In more detail, the rule says: take a string from $\mathcal{L}\langle \textit{BinaryNumeral} \rangle$, follow it with a string from $\mathcal{L}\langle \textit{BinaryDigit} \rangle$, and the result will be an acceptable binary numeral.

What does this mean in practice? Well, we know that `0` is a digit, so our first rule for numerals tells us this digit is also a numeral; the second rule tells us that we can follow this numeral `0` with a digit (either `0` or `1`), and the result will be a numeral: `00` and `01` are therefore both numerals. Similarly, because `1` is a digit and therefore a numeral, we can follow it with a digit, so we know that `10` and `11` are also numerals. And again, since `00`, `01`, `10` and `11` are numerals, we can follow any of these with either `0` or `1` to get numerals `000`, `001`, `010`, `011`, `100`, `101`, `110`, and `111`. Obviously, we can repeat this again and again to get infinitely many numerals:

$$\mathcal{L}\langle \textit{BinaryNumeral} \rangle \ = \ \{\texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}, \texttt{000}, \texttt{001}, \texttt{010}, \texttt{011}, \texttt{100}, \texttt{101}, \texttt{110}, \ldots\} \ .$$

That '...' at the end there means *carry on, according to the rules*, and if we follow the rules, that gives us everything we want in our language of binary numerals. But just as importantly, *we haven't got too much.* The implicit 'and nothing else' means that the '...' isn't hiding monsters such as `5` or `someRandomString`.

Now that we've specified the syntax of binary numerals, the obvious next step is to take a look at their semantics. And that's just what we'll do — before that, though, try these:

Exercise 2.1 *Specify decimal numerals in BNF.*

Lots of things in programming languages have names (we might even say they *are* names); for example, in Java, packages, classes, interfaces, methods, fields, and variables all have names. Such names are usually referred to as *identifiers*, and many programming languages impose restrictions on what is acceptable as an identifier: for example, they may not begin with a digit, may not contain brackets (square, curly, or round) or any other non-alphabetic character except for dots, `$`s or dashes (remember this is all 'for example' — different programming languages have different definitions of what makes an acceptable identifier). A typical example of an acceptable category of identifiers is strings of 'alpahanumerics' (letters and digits) that begin with a letter rather than a digit.

Exercise 2.2

1. *Define a syntactic category, ⟨Alphabetics⟩, that consists of strings of letters, which can be either lower- or upper-case.*
2. *Define a syntactic category, ⟨AlphaNumerics⟩, that consists of strings of digits and letters.*
3. *Define a syntactic category, ⟨Identifier⟩, that consists of strings of letters and digits, that begin with a letter rather than a digit.*

Exercise 2.3 *Another way of writing numbers is*

$$\texttt{0},\ \texttt{0+1},\ \texttt{0+1+1},\ \texttt{0+1+1+1},\ \ldots \tag{2.3}$$

you get the picture; note, though, that `0` *only occurs once, at the left. Specify this in BNF.*

Exercise 2.4 *Your answer to the previous exercise probably wasn't this (even though it's a correct answer):*

⟨*Op*⟩ ::= `+`
⟨*Zero*⟩ ::= `0`
⟨*One*⟩ ::= `1`
⟨*Another Way Of Writing Numbers*⟩ ::= ⟨*Zero*⟩
| ⟨*Another Way of Writing Numbers*⟩⟨*Op*⟩⟨*One*⟩

1. *Why is it obvious that this* is *a correct answer (i.e., that it generates the same language as your answer to the previous exercise)?*
2. *Why not do something even weirder? We could specify* `+`*-linked chains of* `1`*s (the non-zero, or 'positive' numbers) and start off with*

   ⟨*Yet Another Way Of Writing Numbers*⟩ ::=
   ⟨*Zero*⟩ | ⟨*Zero*⟩`+`⟨*Non-zero*⟩

   Specify ⟨*Non-zero*⟩ *so that tiee consists of the strings* `1, 1+1, 1+1+1, ....`
3. *We could even have*

   ⟨*And Yet Another Way Of Writing Numbers*⟩ ::=
   ⟨*Zero*⟩ | ⟨*Non-zero*⟩

   The set of strings generated by this BNF grammar is different to (2.3): how, and why?

Interlude. Since we're talking about different grammars giving the same language, suppose we'd given the rule

⟨*BinaryNumeral*⟩ ::= ⟨*BinaryDigit*⟩ ⟨*BinaryNumeral*⟩

instead of

⟨*BinaryNumeral*⟩ ::= ⟨*BinaryNumeral*⟩ ⟨*BinaryDigit*⟩

This doesn't change the language of binary numerals: we still get

$$\mathcal{L}\langle \textit{BinaryNumeral} \rangle \ = \ \{\texttt{0}, \texttt{1}, \texttt{00}, \texttt{01}, \ldots\} \ ,$$

but how would you explain *why* this is the case? (We'll see how to prove this sort of thing later on, when we look at induction.)

2.1.2 A Little Bit of Semantics

Digits and numerals are strings, sequences of symbols written down in a particular order, in the way specified by the following grammar.

⟨*BinaryDigit*⟩ ::= `0` | `1`
⟨*BinaryNumeral*⟩ ::= ⟨*BinaryDigit*⟩ | ⟨*BinaryNumeral*⟩ ⟨*BinaryDigit*⟩

Just as a decimal numerals work by specifying powers of ten, binary numerals work by specifying powers of two. The binary numeral `1101` represents the number 13, since
draft note: use tables

$$\begin{aligned} 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 &= 1{\times}8 + 1{\times}4 + 0{\times}2 + 1{\times}1 \\ &= 8 + 4 + 0 + 1 \\ &= 13\ . \end{aligned}$$

Similarly, the decimal numeral `1207` represents the number ... well, 1207, since

$$\begin{aligned} 1\times 10^3 + 2\times 10^2 + 0\times 10^1 + 7\times 10^0 &= 1{\times}1000 + 2{\times}100 + 0{\times}10 + 7{\times}1 \\ &= 1000 + 200 + 0 + 7 \\ &= 1207\ . \end{aligned}$$

This representational scheme is called 'the place system', because the value represented by a digit depends upon its place within a numeral: the first `1` in `1101` represents 8, whereas the second `1` represents 4 and the final `1` represents 1 (but in the place system, `0` always represents 0).

So, numerals represent numbers: this is their semantics. The meaning of `1101` is 13, or we might say the numeral `1101` *denotes* the number 13. Let's define a function that takes a numeral as input and returns the number that it denotes. It's traditional in semantics to use double square brackets for denotation-functions like this: we'll write ⟦ `1101` ⟧ for the number denoted by the numeral `1101` (which should be 13). The input to this function (written inside the double brackets) should be a binary numeral, and the result will be a number. It will be useful to introduce a standard notation that specifies what sort of thing the input to a function is, and what sort of thing the output is. So let's write

$$⟦\ \_\ ⟧\ :\ \mathcal{L}⟨\mathit{BinaryNumeral}⟩ \to \mathit{Integer}$$

to indicate that our denotation function (⟦ _ ⟧) takes a binary numeral as input and returns an integer as output. In fact, it will be simpler to define two denotation functions, one for binary digits, and one for binary numerals:

$$\begin{aligned} ⟦\ \_\ ⟧_{\mathrm{D}} &: \mathcal{L}⟨\mathit{BinaryDigit}⟩ \to \mathit{Integer} \\ ⟦\ \_\ ⟧_{\mathrm{N}} &: \mathcal{L}⟨\mathit{BinaryNumeral}⟩ \to \mathit{Integer} \end{aligned}$$

The first of these is straightforward to define, as there are only two possible inputs, `0` and `1`. The digit `0` denotes the number 0:

$$⟦\ \texttt{0}\ ⟧_{\mathrm{D}} = 0$$

and the digit `1` denotes the number 1:

$$⟦\ \texttt{1}\ ⟧_{\mathrm{D}} = 1\ .$$

Interlude.[circularity] Yes, this seems odd: it looks like we're saying nothing at all, and saying nothing-at-all in a complicated notation.

good place to mention Swift's academicians?
and Dijkstra walking out (`x` = true)?

This defines the function $[\![\_]\!]_{\mathrm{D}}$ by *exhaustion*: a rather old-fashioned but useful term that means to go through all possibilities. There are two possible inputs to the function, and we say what the output is in each case. We could do the same for numerals, but because there are an infinite number of numerals, it would take an infinite amount of paper to write it all down. However, there is another way to view this definition of the denotation function for digits: the BNF grammar gives two possible forms for digits:

$$\langle \mathit{BinaryDigit} \rangle \ ::= \ \mathtt{0} \mid \mathtt{1}$$

and our definition of $[\![\_]\!]_{\mathrm{D}}$ follows this pattern — we say what the output of the function is for each of these two possibilities. The BNF grammar for numerals also gives two possible forms for numerals

$$\langle \mathit{BinaryNumeral} \rangle \ ::= \ \langle \mathit{BinaryDigit} \rangle \mid \langle \mathit{BinaryNumeral} \rangle \, \langle \mathit{BinaryDigit} \rangle$$

so can we define $[\![\_]\!]_{\mathrm{N}}$ by exhaustion on just these two possible forms? Yes.

The BNF grammar tells us that a numeral is either a digit or a numeral followed by a digit: if the numeral that is input to $[\![\_]\!]_{\mathrm{N}}$ is a digit, then we can just use $[\![\_]\!]_{\mathrm{D}}$ to work out what its value should be; otherwise, the numeral is a numeral followed by a digit, and we have to do something else....

Let's look at the first possibility: the input is a digit. This digit might be `0` or it might be `1`. Whatever it is, let's just call it D and say

$$\text{for any digit } D, \qquad [\![D]\!]_{\mathrm{N}} \ = \ [\![D]\!]_{\mathrm{D}} \ . \tag{2.4}$$

By giving the name D to the input, we cover both possibilities $D = \mathtt{0}$ and $D = \mathtt{1}$, and we get the desired result in both cases. From (2.4), it follows that

$$[\![\mathtt{0}]\!]_{\mathrm{N}} \ = \ [\![\mathtt{0}]\!]_{\mathrm{D}} \ = \ 0$$

and

$$[\![\mathtt{1}]\!]_{\mathrm{N}} \ = \ [\![\mathtt{1}]\!]_{\mathrm{D}} \ = \ 1$$

and so (2.4) tells us what the output should be in both cases.

Now, if the input isn't a digit, then the 'nothing else' of BNF implies it must be a numeral followed by a digit:

$$\langle \mathit{BinaryNumeral} \rangle \ ::= \ \langle \mathit{BinaryNumeral} \rangle \, \langle \mathit{BinaryDigit} \rangle$$

so the input must have the form ND, where N is a numeral and D is a digit (for example, if the input is `1101`, then N would be `110` and D would be `1`). To complete our definition of $[\![\_]\!]_{\mathrm{N}}$, we need to supply a right-hand side for

$$\text{for any numeral } N \text{ and digit } D, \qquad [\![ND]\!]_{\mathrm{N}} = \ldots \ . \tag{2.5}$$

But what should the right-hand side be?

Have a think.

If you think you know the answer, go straight on[1] to equation (2.7); otherwise, we invite you to try

Exercise 2.5 *You did do Exercise 2.1, yes?*
Good.

And your answer was along the lines of

$$\langle \mathit{DecimalNumeral} \rangle \ ::= \ \langle \mathit{DecimalDigit} \rangle \ \mid \ \langle \mathit{DecimalNumeral} \rangle \, \langle \mathit{DecimalDigit} \rangle$$

yes?
Good.

So, to give a semantics to decimal numerals, you could define two functions

$$\begin{array}{rcl} [\![\ \_ \ ]\!]_D & : & \mathcal{L}\langle \mathit{DecimalDigit} \rangle \to \mathit{Integer} \\ [\![\ \_ \ ]\!]_N & : & \mathcal{L}\langle \mathit{DecimalNumeral} \rangle \to \mathit{Integer} \ . \end{array}$$

Define them.

[1] well, it might be useful to try the exercise anyway

So, we were defining our denotation function: single digits were easy, but we needed to say what to do with larger numerals that were made by sticking a least significant digit (let's call it D) on to the end of a smaller numeral (let's call that N). That is, we needed to complete

$$\text{for any numeral } N \text{ and digit } D, \qquad [\![\, ND \,]\!]_{\mathrm{N}} = \ldots \ . \tag{2.6}$$

The least-significant digit D tells us how many units to add on (either 0 or 1), but what are we going to do with the N? For example, if we're given the numeral `1101`, the least significant digit (`1`) tells us that we need to add 1 to whatever the value of `1100` is, but that value is twice the value of the value of `110` (in binary notation, multiplying by 2 means sticking a `0` on the end). So we have

$$[\![\, \texttt{1101} \,]\!]_{\mathrm{N}} \ = \ [\![\, \texttt{1100} \,]\!]_{\mathrm{N}} + [\![\, \texttt{1} \,]\!]_{\mathrm{D}} \ = \ 2 \times [\![\, \texttt{110} \,]\!]_{\mathrm{N}} + [\![\, \texttt{1} \,]\!]_{\mathrm{D}} \ ,$$

which tells us how to compute the value of a numeral from the value of the least-significant digit and the value of the numeral without that digit. Generalising this gives us:

$$\text{for any numeral } N \text{ and digit } D, \qquad [\![\, ND \,]\!]_{\mathrm{N}} \ = \ 2 \times [\![\, N \,]\!]_{\mathrm{N}} + [\![\, D \,]\!]_{\mathrm{D}} \ . \tag{2.7}$$

For example,

$$\begin{aligned}
[\![\, \texttt{101} \,]\!]_{\mathrm{N}} &= 2 \times [\![\, \texttt{10} \,]\!]_{\mathrm{N}} + [\![\, \texttt{1} \,]\!]_{\mathrm{D}} \\
&= 2 \times [\![\, \texttt{10} \,]\!]_{\mathrm{N}} + 1 \\
&= 2 \times (2 \times [\![\, \texttt{1} \,]\!]_{\mathrm{N}} + [\![\, \texttt{0} \,]\!]_{\mathrm{D}}) + 1 \\
&= 2 \times (2 \times [\![\, \texttt{1} \,]\!]_{\mathrm{N}} + 0) + 1 \\
&= 2 \times (2 \times [\![\, \texttt{1} \,]\!]_{\mathrm{D}} + 0) + 1 \\
&= 2 \times (2 \times 1 + 0) + 1 \\
&= 5 \ .
\end{aligned}$$

Exercise 2.6 *Calculate the values of*

1. $[\![\, \mathit{111} \,]\!]_N$
2. $[\![\, \mathit{1011} \,]\!]_N$
3. $[\![\, \mathit{10100} \,]\!]_N$

2.1.3 Mixing Symbols and Syntactic Categories

---

does this need to be a subsection?

So far, the stuff-following-the-symbol-'::=' in our examples hasn't been particularly complex. We've either had symbols (like `0` and `1`), or syntactic categories, but never a mixture of the two. So let's mix things up, and look at *expressions* that involve numerals and the operation `+`. Intuitively, expressions are sums involving numerals; less intuitively, perhaps, they're strings of numerals separated by `+`-symbols:

⟨*Expression*⟩ ::= ⟨*BinaryNumeral*⟩ | ⟨*Expression*⟩ `+` ⟨*Expression*⟩

This means that if we want to write down an expression, we can either write down a binary numeral (and our earlier BNF grammar tells us how to do that); or we can write down an expression, follow that with the symbol `+`, and then write down another expression. For example, we know that `1101` is a numeral, and therefore it's also an expression; similarly, `11` is a numeral and also an expression, so

```
1101 + 11
```

is an expression. And since `100` and `1` are both numerals, it follows that `100 + 1` is an expression, and so too is

```
1 + 100 + 1
```

and also

```
1101 + 11 + 1 + 100 + 1
```

— we could go on and on and on and on and on....

Exercise 2.7 *Define a denotation function* $[\![\_]\!]_{Exp}$ *that takes an expression as input and gives as result the number that the expression represents. For example,* $[\![\texttt{1101 + 11}]\!]_{Exp}$ *should give the result 16, since this is the sum of 13 (*$[\![\texttt{1101}]\!]_N$*) and 3 (*$[\![\texttt{11}]\!]_N$*).*

Maybe you found that exercise simple. Maybe you looked at the BNF rule

⟨*Expression*⟩ ::= ⟨*BinaryNumeral*⟩ | ⟨*Expression*⟩ `+` ⟨*Expression*⟩

and thought: okay, an expression is either a binary numeral, or it's an expression followed by a `+` symbol, followed by another expression, so any expression is either

- of the form N, where N is a numeral, or
- of the form E_1 `+` E_2, where E_1 and E_2 are expressions,

so I need to find right-hand sides for the equations

$$\text{for all numerals } N, \qquad [\![N]\!]_{\text{Exp}} = \ldots \tag{2.8}$$

$$\text{for all expressions } E_1 \text{ and } E_2, \qquad [\![E_1 \texttt{ + } E_2]\!]_{\text{Exp}} = \ldots . \tag{2.9}$$

then maybe you remembered binary numerals and digits, and completed (2.8) with

$$\text{for all numerals } N, \quad [\![N]\!]_{\text{Exp}} = [\![N]\!]_{\text{N}}$$

and then maybe you hesitated over (2.9) and thought surely it can't be

$$\text{for all expressions } E_1 \text{ and } E_2, \quad [\![E_1 \texttt{ + } E_2]\!]_{\text{Exp}} = [\![E_1]\!]_{\text{Exp}} + [\![E_2]\!]_{\text{Exp}} \ ?$$

But it is. That equation says that the *symbol* `+` is to be interpreted as the operation $+$, addition on numbers

Exercise 2.8 *Go mad: add some more operations to the language of expressions, and repeat the previous exercise, defining a denotation function for the extended language of expressions.*

Sometimes it's useful to include the empty string in a language, for example, if we want to allow an empty list of parameters to a method. Rather than writing "" to indicate the empty string, it's traditional to write ϵ instead (for ϵmpty[2]). For example,

⟨*As*⟩ ::= ϵ | ⟨*As*⟩`a`

describes a language whose strings consist of some number (possibly zero) of `a`s.

Exercise 2.9 *Give a BNF grammar for palindromic strings of* `a`*s and* `b`*s — i.e., the strings contain only the symbols* `a` *and* `b`*, and read the same forwards and backwards, e.g.,* `abba` *and* `babab`*.*

More examples needed here: e.g., binary arithmetic expressions, quadratic equations in x (or something a bit more comp. sci.)

[2]Either that or it stands for 'einheidselement'. This is a German word that's usually translated as 'unity'; if you tried to translate each of the component parts of the word separately, the result would be something like 'one-ness-element', which probably isn't very informative. It does illustrate, though, that maths often works by metaphor and analogy (see the wonderful book by Nuñez and Lakoff [] for much more on this). The idea here is that string concatenation is analogous to multiplication and the empty string is analogous to 1 (the one-ness-element). This analogy is reasonable because multiplying by 1 and concatenating the empty string both have no effect. Take a number, let's call it n, and multiply it by 1 — we write this as $n \times 1$ — and the result is just the number n: in short, $n \times 1 = n$. Now think of n as a string, think of multiplication as string concatenation, and think of the empty string as 1.

2.2 SImpL

SImpL stands for **S**imple **Imp**erative **L**anguage, according to Peter Mosses []. It's a very simple imperative language that we'll use to show how semantics works. We'll restrict the language to the bare bones — assignments to variables, conditionals, and loops — and once we're familiar with those bones we can start hanging some more flesh on them: we'll look at exceptions, and other stuff in exercises. For the moment, though, we'll concentrate on the syntax of our basic language

principle: some kinds of thing can be used in some contexts, but not in others — syntactic categories need to distinguish these. E.g., `x = 2 * x` is okay in Java, but not `2 * x = x`, so distinguish variables and expressions.

⟨*Variable*⟩ ::= `'a` | `'b` | ··· | `'aa` | `'ab` | ···

⟨*Assignable*⟩ ::= ⟨*Variable*⟩

⟨*Expression*⟩ ::= ⟨*Assignable*⟩ | ⟨*Numeral*⟩
| `-` ⟨*Expression*⟩
| ⟨*Expression*⟩ `+` ⟨*Expression*⟩
| ⟨*Expression*⟩ `-` ⟨*Expression*⟩
| ⟨*Expression*⟩ `*` ⟨*Expression*⟩

⟨*BooleanExpression*⟩ ::= `true` | `false`
| ⟨*BooleanExpression*⟩ `&&` ⟨*BooleanExpression*⟩
| ⟨*BooleanExpression*⟩ `||` ⟨*BooleanExpression*⟩
| `!` ⟨*BooleanExpression*⟩
| ⟨*Expression*⟩ `<` ⟨*Expression*⟩
| ⟨*Expression*⟩ `==` ⟨*Expression*⟩

⟨*Program*⟩ ::= `skip`
| ⟨*Assignable*⟩ `:=` ⟨*Expression*⟩`;`
| ⟨*Program*⟩ ⟨*Program*⟩
| `if (`⟨*BooleanExpression*⟩`) {` ⟨*Program*⟩ `} else {` ⟨*Program*⟩ `}`
| `while (`⟨*BooleanExpression*⟩`) {` ⟨*Program*⟩ `}`

2.3 Interlude: Induction

Remember we said that every BNF grammar comes with an implicit closure: 'nothing else belongs to the language'? This reassures us that the language defined by the grammar consists of only the strings allowed by the rules. This means that the language doesn't hold any surprises, so we can define functions like the denotation of digits and numerals by induction on the possible forms that the strings of the language can take. It turns out that this closure condition has some very useful consequences that allow us to reason about the language.

2.3.1 Adding one: an exercise in syntactic manipulation

Try this exercise: you know how to add one to any number; but can you *explain* how to do it? To take you out of your comfort zone, we'll ask you to do it in base 2.

Exercise 2.10 *Define a function that takes as input a binary numeral and returns the numeral that is one greater than the given numeral.*

A solution is given below, so try the exercise now.

You were asked to define an increment function on binary numerals: for any binary numeral N, give the numeral whose value is one greater than the value of N. To put this in formal notation, you were asked to

define a function

$$inc \ : \ \mathcal{L}\langle \mathit{BinaryNumeral} \rangle \to \mathcal{L}\langle \mathit{BinaryNumeral} \rangle$$

such that for any binary numeral N,

$$[\![\ inc(N)\ ]\!]_{\mathrm{N}} \ = \ [\![\ N\ ]\!]_{\mathrm{N}} + 1 \ . \tag{2.10}$$

You probably came up with a definition that looked like this:

$$\begin{aligned}
inc(\mathtt{0}) &= \mathtt{1} && && (2.11)\\
inc(\mathtt{1}) &= \mathtt{10} && && (2.12)\\
inc(M\mathtt{0}) &= M\mathtt{1} && \text{for any } M \in \mathcal{L}\langle \mathit{BinaryNumeral} \rangle && (2.13)\\
inc(M\mathtt{1}) &= inc(M)\mathtt{0} && \text{for any } M \in \mathcal{L}\langle \mathit{BinaryNumeral} \rangle && (2.14)
\end{aligned}$$

and if you did, you're probably confident that it's right: the first three equations speak for themselves, and the final equation is just the 'carry one' rule that you learnt years ago in school (though maybe not for binary numerals). But suppose you wanted a more formal argument to show that your solution is correct — how could you persuade even the most cynical doubter that you've given a correct method for adding one to a binary numeral? One obvious way would be to show that (2.10) holds for all possible values of N; we could show that

$$\begin{aligned}
[\![\ inc(\mathtt{0})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{0}\ ]\!]_{\mathrm{N}} + 1\\
[\![\ inc(\mathtt{1})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{1}\ ]\!]_{\mathrm{N}} + 1\\
[\![\ inc(\mathtt{10})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{10}\ ]\!]_{\mathrm{N}} + 1\\
[\![\ inc(\mathtt{11})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{11}\ ]\!]_{\mathrm{N}} + 1\\
[\![\ inc(\mathtt{100})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{100}\ ]\!]_{\mathrm{N}} + 1\\
[\![\ inc(\mathtt{101})\ ]\!]_{\mathrm{N}} &= [\![\ \mathtt{101}\ ]\!]_{\mathrm{N}} + 1\\
&\ldots
\end{aligned}$$

but we've been here before. This is exactly the same problem we had in defining $[\![\ N\ ]\!]_{\mathrm{N}}$: there are an infinite number of possibilities for N, and we just don't have time to go through them all. The solution happens to be the same.

When you defined $inc(N)$ using the four equations above, you were looking at the four possible forms that the (arbitrary) numeral N could take. Either:

1. N is a digit, so either
   (a) $N = \mathtt{0}$, or
   (b) $N = \mathtt{1}$; or
2. $N = M\mathtt{0}$ for some M; or
3. $N = M\mathtt{1}$ for some M.

So, instead of trying to show that (2.10) holds for all the possible values that N might take (i.e., each and every binary numeral), you might try instead to show that it holds in each of these four possible cases. For the first case, we can calculate

$$[\![\ inc(\mathtt{0})\ ]\!]_{\mathrm{N}} \ = \ [\![\ \mathtt{1}\ ]\!]_{\mathrm{N}} \ = \ 1$$

(the first step uses (2.11) from the definition of inc) and then check that this equals $[\![\ \mathtt{0}\ ]\!]_{\mathrm{N}} + 1$, which shows (2.10) for the case 1a, where $N = \mathtt{0}$. That's a good start.

Exercise 2.11 *Look back at (2.10); check that this really is an instance of that equation.*

Doing the same thing for the second case, we get

$$[\![\ inc(\mathtt{1})\ ]\!]_{\mathrm{N}} \;=\; [\![\ \mathtt{10}\ ]\!]_{\mathrm{N}} \;=\; 2 \;=\; [\![\ \mathtt{1}\ ]\!]_{\mathrm{N}} + 1 \tag{2.15}$$

which shows (2.10) for the case 1b, where $N = \mathtt{1}$. Things are looking good.

For the third case, where $N = M\mathtt{0}$ for some numeral M, we calculate

$$[\![\ inc(M\mathtt{0})\ ]\!]_{\mathrm{N}} \;=\; [\![\ M\mathtt{1}\ ]\!]_{\mathrm{N}} \;=\; 2 \times [\![\ M\ ]\!]_{\mathrm{N}} + 1 \;=\; [\![\ M\mathtt{0}\ ]\!]_{\mathrm{N}} + 1 \tag{2.16}$$

and this shows (2.10) for case 2 — we're on a roll!

Let's pause here to try to build up some dramatic tension. We've defined the function *inc* and we *know* we've defined it correctly. The function is defined by equations (2.11)–(2.14) and this definition follows the definition of binary numerals

$$\langle \mathit{BinaryNumeral} \rangle \;::=\; \langle \mathit{BinaryDigit} \rangle \;\mid\; \langle \mathit{BinaryNumeral} \rangle\, \langle \mathit{BinaryDigit} \rangle$$

If we factor in the fact that each binary digit is either `0` or `1`, the definition is

---

this refactorisation should be mentioned earlier: more than one way to skin a cat

---

$$\langle \mathit{BinaryNumeral} \rangle \;::=\; \mathtt{0} \;\mid\; \mathtt{1} \;\mid\; \langle \mathit{BinaryNumeral} \rangle\, \mathtt{0} \;\mid\; \langle \mathit{BinaryNumeral} \rangle\, \mathtt{1}$$

However, when we look at the case where $N = M\mathtt{1}$ for some numeral M, then we get a bit stuck:

$$[\![\ inc(M\mathtt{1})\ ]\!]_{\mathrm{N}} \;=\; [\![\ inc(M)\mathtt{0}\ ]\!]_{\mathrm{N}} \;=\; 2 \times [\![\ inc(M)\ ]\!]_{\mathrm{N}} \;=\; \ldots ?$$

There's nothing more we can do here; we can't simplify $[\![\ inc(M)\ ]\!]_{\mathrm{N}}$ any further. The problem is that we don't know what the numeral M is: 'M' is just a name standing for an arbitrary numeral. If we take M to be `1`, for example, then we could continue

$$2 \times [\![\ inc(M)\ ]\!]_{\mathrm{N}} \;=\; 2 \times [\![\ inc(\mathtt{1})\ ]\!]_{\mathrm{N}} \;=\; 2 \times [\![\ \mathtt{10}\ ]\!]_{\mathrm{N}} \;=\; 2 \times 2 \;=\; 4\ .$$

In this case, we can just use (2.15): we've already worked out what $[\![\ inc(\mathtt{1})\ ]\!]_{\mathrm{N}}$ is. In fact, if we know that (2.10) holds also for M — that is, if we know that $[\![\ inc(M)\ ]\!]_{\mathrm{N}} = [\![\ M\ ]\!]_{\mathrm{N}} + 1$ — then we can just calculate

$$\begin{aligned} [\![\ inc(M\mathtt{1})\ ]\!]_{\mathrm{N}} \;&=\; [\![\ inc(M)\mathtt{0}\ ]\!]_{\mathrm{N}} \;=\; 2 \times [\![\ inc(M)\ ]\!]_{\mathrm{N}} \;=\; 2 \times ([\![\ M\ ]\!]_{\mathrm{N}} + 1) \\ &=\; 2 \times [\![\ M\ ]\!]_{\mathrm{N}} + 2 \;=\; (2 \times [\![\ M\ ]\!]_{\mathrm{N}} + 1) + 1 \;=\; [\![\ M\mathtt{1}\ ]\!]_{\mathrm{N}} + 1 \end{aligned}$$

and this shows that (2.10) holds for the case where $N = M\mathtt{1}$ for some numeral M. Note that it's the third equality in this chain where we use our assumption that $[\![\ inc(M)\ ]\!]_{\mathrm{N}} = [\![\ M\ ]\!]_{\mathrm{N}} + 1$. What we've just done is an example of an *inductive proof* that (2.10) holds for all numerals N.

You might already be familiar with mathematical induction, in which case, feel free to skip ahead to Section 2.3.3. If you're not familiar with mathematical induction, the following section gives a brief introduction; you might also want to read a more detailed account in, for example, [**?**] or in Chapter 2 of [**?**].

2.3.2 Mathematical Induction: an interlude within an interlude

Numbers are, perhaps, the most sophisticated of all cultural artefacts, painstakingly built up over millenia by several different cultures[3]. Numbers are basically used for counting, and most cultures have words for one and two. For some cultures, that's enough, and two is followed by 'many'.

[3]See [**?**] for a very good overview; [**?**] gives another fascinating account from a slightly different perspective

2.3.3 Induction on Syntax

the interlude on page 12

2.4 Ambiguity

The syntax of SIMPLE allows us to write the expression `1 + 2 * 3`. In fact, there are two different ways of seeing that this is a well-formed expression. We can see from the syntax given on page 20 that `2` and `3` are both numerals; therefore they are also both expressions; and therefore `2 * 3` is an expression. Since `1` is a numeral and therefore also an expression, it follows that `1 + 2 * 3` is an expression. Although we haven't got round yet to looking at the semantics of SIMPLE expressions, we would expect this expression to evaluate to 7. The other way of seeing that the expression is well-formed is to start off by noting that `1` and `2` are both numerals, and therefore both expressions, so `1 + 2` is an expression; 'multiplying' this by three gives us the expression `1 + 2 * 3`. This way of looking at the expression is similar to the bracketed expression `(1 + 2) * 3`, whereas our first way of looking at it corresponds to the expression `1 + (2 * 3)`.

The syntax of SIMPLE is *ambiguous*:

2.5 Some More Exercises

Exercise 2.12 *Look at the definition of ASCII (for example, at http://)*

Example 2.1 *Define increment*

Exercise 2.13 *Define addition on numerals.*

Exercise 2.14 *What is the relationship between exercises 2.7 and 2.13.*

Exercise 2.15 *Here's a more ambitious one. Give a BNF definition of the syntax of BNF grammars; then give a semantics for BNF grammars.*

Chapter 3

Classical Denotational Semantics

Denotational semantics describes the meaning of a program as some mathematical object: this is the program's *denotation*. Typically, the denotation of a program is a function that maps states to states. Given a program P, its denotation, $[\![P]\!]_{\text{Pgm}}$, is the function that takes a state S as argument and maps it to the state that results from running P in the state S, provided that the program P terminates in the state S.

We first give a denotational semantics for arithmetic expressions, then for Boolean expressions, and finally for programs. Before all that, we first define what we mean by a **state**.

3.1 State

In any imperative language, assignment is the most basic form of program: the other linguistic constructs (conditionals and loops) are simply ways of organising assignments into series that are executed in a particular order. For example, think of how (according to our intuitions) the following program is evaluated:

```
'x := 0 ;
'f := 1 ;
while ('x <= 2) {
    'x := 'x + 1 ;
    'f := 'f * 'x ;
}
```

On each pass through the loop, the value of `'x` is increased by 1, and the body of the loop is gone through three times (while `'x` has values `0`, `1`, then `2`). Thus, we could "unfold" the program above to the following series of assignments:

```
'x := 0 ;
'f := 1 ;
'x := 'x + 1 ;
'f := 'f * 'x ;
'x := 'x + 1 ;
'f := 'f * 'x ;
'x := 'x + 1 ;
'f := 'f * 'x ;
```

(one of the benefits of a formal semantics is that we could give a rigorous argument that these two programs are indeed equivalent).

Assignments, therefore, lie at the heart of imperative programming languages. They also lie at the heart of the semantics of imperative languages. In particular, they suggest that any semantics should be based on the notion of *state* (or *storage*), by which we mean the particular values that are associated with a program's variables. When a program is being executed, we can think of the computer that is running the program as being in a certain state. This state is determined by the values stored by the program's variables, and these values are updated by assignments to the variables. For example, after running the program above,

the computer will be in a state where the variable `'x` has the value `3`, and the variable `'f` has the value `6`. We can think of a state as being a table that tells us the value associated with any given variable (in our language, variables aren't declared, and have no scope, so the state should tell us the value associated with *any* variable. This would give us an infinitely long table, and it is more convenient to think of a state as a function that takes a variable as argument and returns the value stored in that variable. More formally, a state is a function $\langle \mathit{Variable} \rangle \to \mathit{Int}$. For example, after running a program

```
'x := 8;
'y := 'x + 1;
'z := 'y + 2;
```

we obtain a state s, with $s(\texttt{'x}) = 8$, $s(\texttt{'y}) = 9$, and $s(\texttt{'z}) = 11$. (We don't know what values s gives to variables other than `'x`, `'y` or `'z`, but presumably those values aren't changed by the program.)

The semantics we will give will describe how assignments (and other programs) update states. Before we do this, note that an assignment like

```
'y := 'x + 1 ;
```

depends on the state of the computer before the assignment is executed: the value assigned to `'y` depends on the value of `'x` in this "initial" state (e.g., if the value of `'x` in the initial state is 8, then `'y` is assigned the value 9, and if `'x` has the value 23, then `'y` is assigned the value 24).

3.2 The Denotational Semantics of Arithmetic Expressions

Given an arithmetic expression such as

```
2 * ('x + 'y)
```

we want to describe its denotation as a number, but clearly the value of such an expression depends on the values stored in the variables `'x` and `'y`. In other words, the value of an arithmetic expression depends on the state of the computer when the expression is evaluated. We therefore describe the denotation of an arithmetic expression e as a function $[\![e]\!]_{\mathrm{Pgm}} : \mathit{State} \to ZZ$. This function is defined inductively as follows:

- For a number n, clearly the value should just be n, independent of the state:

  $$[\![n]\!]_{\mathrm{Pgm}}(s) = n.$$

  (more precisely, we should distinguish between numerals and numbers).

- For a variable x, the value should simply be the value in the given state:

  $$[\![x]\!]_{\mathrm{Pgm}}(s) = s(x).$$

- When e has the form $e1$ `+` $e2$, the value should be the sum of the values of $e1$ and $e2$:

  $$[\![e1\texttt{+}e2]\!]_{\mathrm{Pgm}}(s) = [\![e1]\!]_{\mathrm{Pgm}}(s) + [\![e2]\!]_{\mathrm{Pgm}}(s).$$

- The operations for unary minus and multiplication are treated similarly.

3.3 The Denotational Semantics of Boolean Expressions

This is left as an exercise!

3.4 The Denotational Semantics of Programs

The denotational semantics for programs is given by defining, for a program P, its denotation $[\![P]\!]_{\mathrm{Pgm}}$. From our intuitive understanding of programs (strengthened by having seen the operational semantics), we know that executiong a program has the effect of updating the state in which execution of the program begins. Thus, the denotation $[\![P]\!]_{\mathrm{Pgm}}$ will be a function that takes a state s as argument and returns the state that results from executing P in the state s. Of course, there is the possibility that P fails to terminate when it is executed in s, in which case $[\![P]\!]_{\mathrm{Pgm}}(s)$ shouldn't return any state: in other words, $[\![P]\!]_{\mathrm{Pgm}}(s)$ is undefined. This means that, in general, $[\![P]\!]_{\mathrm{Pgm}}$ is a *partial* function from states to states. (The denotation functions $[\![e]\!]_{\mathrm{Exp}}$ and $[\![t]\!]_{\mathrm{BExp}}$ for expressions and tests are both total functions from states to numbers and truth-values, respectively, as evaluating arithmetic and Boolean expressions always terminates.)

The function $[\![P]\!]_{\mathrm{Pgm}}$ is defined inductively as follows:

- $[\![\texttt{skip}]\!]_{\mathrm{Pgm}}(s) = s$
- $[\![x\texttt{:=}e]\!]_{\mathrm{Pgm}}(s) = s[n/x]$, where $n = [\![e]\!]_{\mathrm{Exp}}(s)$.
- $[\![P_1\ P_2]\!]_{\mathrm{Pgm}}(s) = [\![P_2]\!]_{\mathrm{Pgm}}([\![P_1]\!]_{\mathrm{Pgm}}(s))$.
- $[\![\texttt{if (}T\texttt{) \{}P_1\texttt{\} else \{}P_2\texttt{\}}]\!]_{\mathrm{Pgm}}(s) = [\![P_1]\!]_{\mathrm{Pgm}}(s)$ if $[\![T]\!]_{\mathrm{BExp}}(s) = \mathit{true}$.
- $[\![\texttt{if (}T\texttt{) \{}P_1\texttt{\} else \{}P_2\texttt{\}}]\!]_{\mathrm{Pgm}}(s) = [\![P_2]\!]_{\mathrm{Pgm}}(s)$ if $[\![T]\!]_{\mathrm{BExp}}(s) = \mathit{false}$.
- $[\![\texttt{while (}T\texttt{) \{}P\texttt{\}}]\!]_{\mathrm{Pgm}}(s) = s$, if $[\![T]\!]_{\mathrm{BExp}}(s) = \mathit{false}$.
- $[\![\texttt{while (}T\texttt{) \{}P\texttt{\}}]\!]_{\mathrm{Pgm}}(s) = [\![\texttt{while (}T\texttt{) \{}P\texttt{\}}]\!]_{\mathrm{Pgm}}([\![P]\!]_{\mathrm{Pgm}}(s))$, if $[\![T]\!]_{\mathrm{BExp}}(s) = \mathit{true}$.

Chapter 4

Maude

in which we learn a means of describing many structures of things

---

4.1 What is Maude?

Maude is a language for specifying abstract data types. An abstract data type consists of a set of data values, together with operations that work on those data values; accordingly, Maude allows users to specify an abstract data type by saying what the data values are, and by saying what the operations are, and how they behave.

4.2 Abstract Data Types

Computer programs work with data. The data that a program works with typically represents some information about the real world. For example, a company that sells CDs might use a program that uses a database to store details about the CDs they have in stock, and allows users (perhaps through a web page) to place orders for particular CDs. Within such a program, a database entry might look like (this is a hypothetical example, so we'll make up an *ad hoc* notation):

$$\langle\ \textbf{key} : \text{ARISTA 37927},\ \textbf{title} : \text{Horses},\ \textbf{artist} : \text{Patti Smith},\ \textbf{qty} : 257\ \rangle\ .$$

This data might represent the information that the company has in stock 257 copies of a CD called 'Horses' by Patti Smith, with catalogue number ARISTA37927. (Next up in the hierarchy from 'data' and 'information' is *knowledge*; in this case, the knowledge that underlies the information is that this was the first album of an American poet whose music was perhaps better than her poetry.) Similarly, our hypothetical program might use something like (again, we'll make up an *ad hoc* notation):

```
<class="order">
    <class="client">
        <name="G. Malcolm">
        <address="Dept Comp Sci, Univ Liverpool, UK">
    </client>
    <class="item">
        <catalogueNumber="ARISTA~37927">
    </item>
    <qty="1"/>
</order>
```

as data to represent the information that a G. Malcolm has placed an order for one copy of the CD ARISTA 37927. This in turn might represent the real-world *knowledge* (which you may or may not have) that the distraught customer placed the order after his vinyl copy of the original recording was left on the turntable in the full glare of an unusually warm February sun.

As another example, a program for calculating students' overall grades might use a table to collate grades for individual modules. Particular data for two students might look like this (once again, an *ad hoc* notation):

Name	M1	M2	M3	M4	Average
Brill, Sue	75	98	82	68	81
Sole, Pertwee	25	2	17	11	14

. . . and you can probably guess what information might be represented by such data. In fact, the *ad hoc* notations we used in these examples is intended to suggest the information that they represent. A less suggestive notation for the same data presented in the first row of this example might simply be, for example:

$$(\text{“}\texttt{Brill}, \texttt{Sue}\text{”}, 75, 98, 82, 68, 81)$$

— and it would be much harder to guess what information might be represented by that notation for the data (but note that what's changed is just the notation, not the data).

Of course, as far as our hypothetical programs are concerned, the question of notation is irrelevant. The six values printed above might never be written down in any notation at all; they might simply be the values of six variables used in the program (the variables might possibly be fields in an instance of a class `StudentSemesterGradeRecord`), or entries in a database table. But we do want to make two points from these examples:

- programs use data, and the data they use tends to be *structured*: values are grouped together in ways that reflect the information that relates those values in the real world (or at least in some abstraction of the real world: what is usually called the *domain of application*), and

- as soon as we talk about — or write about — the data used by a program, it's useful to have some notation that reflects the structure of the data; for intelligibility, it helps if that notation suggests the information that groups the data into its structure.

We'll see that Maude allows us to address both these points, by allowing us to specify notation for structures of data.

We still haven't addressed one vital point, though: the reason we want to group data in a structured way is that we want to write programs that will do useful things with that data. We might, for example, want to write programs that will test whether a particular order for a particular CD can be met from the CDs that are currently in stock, and if so, create a request to ship that CD from a particular warehouse, print off shipping and billing information, and so on. In order to specify what we want programs to do, then, we need to specify: what data a program will work with; how that data is structured; and what we want the program to do with that data.

One final point: the reason for *specifying* data structures rather than just going ahead and writing the programs that use those data structures is that writing programs is easy, but knowing *what* program to write is fiendishly difficult. To put it another way: if you know what your program should do, writing that program is fairly straightforward; it's knowing what the program should do that is often the tricky part.

Summary

An abstract data type (ADT) is data that is structured in some coherent and useful way, and that can be worked upon in systematic ways. Examples are: pairs, lists, stacks, binary trees, strings, . . . and lots more. To say that the data is *structured* might mean that values are organised in pairs, or sequences, or separated into left and right subtrees, and so on. To say that there are systematic ways of working with the data means that there are specific operations that do everything we want to do with the data (at least for some specific application).

Background

The term 'abstract data type' seems to have originated with Parras (1976)....

Specification languages: CLEAR, OBJ, Maude, CASL.... Prevalence in functional languages, e.g., Haskell (algebraic data types and work of Ghani)....

Meyer: OO classes and ADTs....

An Abstract Data Type (ADT) consists of:

- a set of abstract data values; and
- some operations that act upon those values.

4.3 Maude and Abstract Data Types

Let's start with a very simple example of an abstract data type: booleans. There are just two boolean values: *true* and *false*. That is, the set of abstract data values for this ADT is just

$$\{true, false\} .$$

Let's call this set *Bool*. What operations do we want for working with these values? — well, we're specifying the ADT, so we can choose whatever we want. To keep it simple, let's just have negation (*not*) and conjunction (*and*). And that's it. Well, almost; if we want to give a precise description of this ADT, we should also say:

- how many arguments these operations take (*not* is unary: it takes just one argument, and *and* is binary: it takes two), and what type those arguments are (all arguments should be from the set *Bool*),
- what type of results these operations give (*not* takes one argument of type *Bool* and gives a result of type *Bool*; *and* takes two arguments of type *Bool* and gives a result of type *Bool* as well),
- and what exactly the operations *do*....

The operations *not* and *and* are *functions*: we specify what they *do* by saying what the output is for all the possible inputs. We might do this by going through all the possible inputs; for example, we can define *not* inductively by saying that *not*(*true*) is *false* and *not*(*false*) is *true*. Another way of doing the same thing is to use a truth table; for example, *and* is defined by the following table.

X	Y	X *and* Y
true	*true*	*true*
true	*false*	*false*
false	*true*	*false*
false	*false*	*false*

It's turned out to be quite complex to specify the booleans with just two operations. Things might have gone more smoothly if we'd had a dedicated notation for specifying sets of abstract data values, operations, and how they behave. Maude is exactly such a notation. Here's what the ADT of booleans looks like when specified in Maude:

```
fmod BOOLEANS is

  sort Bool .

  ops true false : → Bool .
  op not : Bool → Bool .
  op _and_ : Bool Bool → Bool .

  eq not(true) = false .
```

```
  eq   not(false) = true .

  var  Y : Bool .

  eq   true and Y = Y .
  eq   false and Y = false .

endfm
```

Let's look more closely at the various parts of this specification.

Maude Modules

Maude specifications are *modular*: each specification is contained in a 'module'. There are different types of modules, but we need only concern ourselves with one of these: BOOLEANS is an example of what's called a 'functional module' — this is indicated by the keywords **fmod** and **endfm** that begin and end the module. In order to understand the module above, we need to know that:

- modules have names, and
- inside modules, we put *declarations*; these can declare:
  - sorts,
  - operations,
  - variables and equations.

We'll have a closer look at each of these.

Module Names

Booleans are useful in specifying other data types: for example, when specifying Stacks, we might want an operation that tests whether a given stack is empty, i.e., this operation should return a Bool. In order to use one module inside another, without copying the entire module, we can refer to it by name. The name of our example module is BOOLEANS, and in general, the name of a module comes between the keywords **fmod** and **is**. A name is just a string of letters with no spaces, and we can choose any name we like. Of course, it's a good idea to choose a meaningful name. It's also a good idea to adopt a convention for naming things: we'll follow the standard Maude practice of writing names of modules in ALL-CAPITALS, with hyphens separating any words (underscores are often commonly used). This convention isn't enforced in any way (the Maude interpreter won't give an error message if it comes across a module name that isn't in ALL-CAPS), but it means that module names will be immediately recognisable as such.

Following the keyword **is**, and before the keyword **endfm**, is the meat of the specification, the declarations of sorts, operations, variables and equations. Here is the simplest, and quite possibly the strangest Maude specification you'll ever see, a vegetarian specification with no declarations:

```
fmod NOTHING is

endfm
```

It's hard to think of this as a specification at all. There's no set of abstract data values, and no operations on ... — well, there's nothing to operate on! This is an extreme (and, frankly, disturbing) example of what you can write in Maude.

The second-simplest Maude specification is also a bit strange, but at least it introduces our next topic.

```
fmod ZERO is

  sort Nothing .

endfm
```

Sorts

Sort declarations are introduced by the keyword **sort**, and what they do is introduce a name for a set of abstract data values. For example,

```
sort Bool .
```

simply introduces the name 'Bool' for the set of abstract data values in our specification of booleans. That's it: just a name. We'll see later on that the abstract data values themselves will be given by the *terms* that we can write using the operations we declare. The name of the sort is used to specify the *types* of operations: the types of their arguments and the type of value they return (the designers of Maude chose the word 'sort' because they felt the word 'type' had too many different meanings in Computer Science).

Note that another Maude keyword appears in sort declarations: the full-stop (**.**) at the end. All the different kinds of declarations in Maude begin with a keyword and end with a full-stop — and this full-stop *must* be preceded by a space[1] (which probably makes the space another Maude keyword).

Again, the name of the sort can be anything you like, but — again — it's a good idea to choose meaningful names, and to follow the Maude convention of beginning a sort name with a capital letter. This also happens to be the convention for naming classes in Java, and if you want a name that contains several words, you can squash them all up into one word and begin each component word with a capital letter, for example, BankAccount, BinaryTree, etc. As with module names, this convention isn't enforced in any way, but it does mean you can spot sort-names instantly [2].

If you're writing a specification that declares several sorts, you can declare them in one go using the keyword **sorts**. For example,

```
sorts Bool Int BinaryTree .
```

is equivalent to

```
sort Bool .

sort Int .

sort BinaryTree .
```

Operations

Sort declarations introduce names for sets of abstract data values; operation declarations introduce names for operations that work with those values, and also specify what sorts of arguments the operations take, and what sort of values they return. The keyword **op** introduces an operation declaration; it is followed by the name of the operation, which is followed by the keyword **:** (a colon), which is followed by a list of sort names that says what sorts of arguments the operation takes, which is followed by the keyword → (in ASCII, this is a minus followed by a greater-than: `->`), which is followed by the name of the sort that the operation returns . . . and then the declaration is ended with the space-full-stop keyword. Sounds complicated, doesn't it? But really, it's just: name, argument sorts, return sort. For example,

```
op not : Bool → Bool .
```

[1] Unless the character that precedes the full-stop is a 'special character'; i.e., a bracket (round, square, or curly).

[2] This is especially useful when you make a mistake and mis-type a name in a program; if the error message says the name 'sortOfThing' is unrecognised, the lower-case letter at the start jumps out.

declares an operation called 'not', which takes one argument of sort Bool, and returns a Bool. Note that the list of argument sorts are just separated by spaces, for example,

```
op  and : Bool Bool → Bool .
```

declares an operation that takes two arguments, both of sort Bool, and returns a Bool. In our example, we only have one sort, so all the argument sorts and return sorts are going to be the same, but if we were specifying balanced binary trees, we might have

```
sorts Bool Int BalancedBinaryTree .

op  insert : Int BalancedBinaryTree → BalancedBinaryTree .

op  isEmpty : BalancedBinaryTree → Bool .
```

There is an interesting special case of operation: operations that take *no* arguments. We call these *constants*. For example, true is a Boolean value — it just is, and doesn't need any input. No inputs are written in Maude as nothing between the colon ':' and the →. For example

```
op  true :  → Bool .
```

Similarly, if we wanted to specify binary digits, we'd need to declare a sort BinaryDigit:

```
sort  BinaryDigit .
```

and two constants, 0 and 1. (They're constants because they simply are digits and don't require any input/arguments.) So we specify these constants as follows.

```
op  0 :  → BinaryDigit .
op  1 :  → BinaryDigit .
```

This is another very simple Maude specification, so it's worthwhile giving it in full.

```
fmod  BINARY-DIGITS  is

  sort  BinaryDigit .
  ops  0 1 :  → BinaryDigit .

endfm
```

Note that we use the keyword **ops** to declare several operations in one go — but we can only do this if all the operations take the same sorts of argument, and have the same return sort.

What about choosing names for operations? Meaningful names are good, of course, and it helps recognise operation names if you follow the convention of beginning them with lower-case letters, as in the examples above. As you might expect by now, this convention is also not enforced, which is a good thing, because sometimes we want to specify an operation that is usually denoted by a symbol rather than a string of letters. For example, if we were specifying numbers, and wanted an operation to add two numbers together, we might want to use the symbol '+' as a name, rather than, say, 'add' or 'plus'. Well, that symbol is as good a name as any other, and we can declare, for example

```
op  + : Int Int → Int .
```

Now, a symbol such as '+' is often used as an *infix* operation: that is, we usually write 2 + 2 rather than +(2, 2). Maude allows users to specify their own syntax by means of underscores. Underscores can be placed inside 'names' of operations, and say where the arguments should go — of course there should be exactly the same number of underscores as there are arguments for the operation. For example, we can declare an infix addition operation as follows.

```
op  _+_ : Int Int → Int .
```

Here, the operation takes two arguments, so we use two underscores. The first argument goes where the first underscore is (on the left of '+'), and the second argument goes where the second underscore is (on the right). Thus, if we were to apply this operation to arguments 12 and 47, we would write it as '12 + 47'. As another example, the factorial operation is usually written as a postfix exclamation mark. We can declare this syntax in Maude as follows.

```
op  _! : Int → Int .
```

And this allows us to write 24! for the factorial of 24.

In fact, underscores allow us to write any kind of 'form' for applying operations to arguments. Suppose we wanted to specify pairs of integers, and suppose we really want to write pairs using a notation like `< 2, 26 >`. Then we might have a Maude specification containing the following declarations:

```
sort Pair .
op  ⟨_,_⟩ : Int Int → Pair .
```

Similarly, if we think of a while-loop as an operation that takes a guard and a body as arguments, and returns the while-loop itself as result, then we can specify our desired syntax as follows:

```
op  while_do_od : BooleanExpression Program → Program .
```

And in fact, that's exactly what we will do later on.

The remaining kinds of declarations that can go into a Maude specification are variables and equations, but before we go on to look at those, let's take another look at the simplest kind of operation, the ones that don't take any arguments at all: the **constants**. We saw two of these in BOOLEANS:

```
ops  true false :  → Bool .
```

Note that the list of argument sorts, between the '**:**' and the '→', is empty. These operations don't take any arguments, they simply *are* boolean values.

The keyword **ops** is to **op** as **sorts** is to **sort**: it allows more than one operation to be declared in one go. However, all the declared operations must have the same type: they must take the same number of arguments, all of the same sorts (that is, the lists of argument sorts must be the same, not that all the sorts in the list must be the same), and the return sorts must be the same as well. Thus, the declaration above is equivalent to:

```
op  true :  → Bool .

op  false :  → Bool .
```

Similarly, suppose we wanted to specify bank accounts. We might start off by declaring a

```
sort Account .
```

and then go on to declare two operations: the first operation allows one to withdraw money from the account:

```
op withdraw : Account Int → Account .
```

— given an account and integer as arguments, this operation returns the account where the given integer number of Euros has been withdrawn — and another operation that allows one to deposit money into an account:

```
op deposit : Account Int → Account .
```

This should be the opposite of 'withdraw': given an account A and an integer I, deposit(A,I) represents the same account as A, but with the difference that it contains I (the amount deposited) more Euros. Since the types of these two operations are the same, we can conflate these two declarations into one:

```
ops withdraw deposit : Account Int → Account .
```

But note that we can't write 'deposit(A,I)' unless we have an account A or an integer I that we can write down; we can't write 'not(X)' or 'X and Y' unless we have booleans X and Y that we can write down. And that's the wonderful thing about constants: they don't take any arguments, so we can just write true or false, and we've got boolean values. Writing things down is one of the really useful things that Maude allows us to do. We can write *terms*.

Terms

So far, operations don't look that useful. Rather like sorts, they just seem to be names, or just syntax. Remember, though, that operation declarations also give type information: the sorts of the arguments, and the sort of the result. Remember the declaration of negation:

```
op not : Bool → Bool .
```

This says we have an operation called 'not' that takes an argument of sort Bool and returns a Bool. When we don't use underscores, we apply an operation by puting the arguments within brackets, and separated by commas — 'not' only takes one argument, so we don't need commas, but what can we put inside the brackets? This is where constants come in: 'true' doesn't take any arguments, so 'true' by itself is a Bool. So we can apply 'not' to 'true', and we write this down as 'not(true)'. This gives us two examples of *terms*: things we can write down that are well-formed terms of sort Bool. 'Well-formed' means that all operations are applied to the correct number of arguments, and those arguments are of the correct sort. Examples of terms that are *not* well formed are:

not	because not requires an argument
not(true, true)	because not requires one and not two arguments
true and	because and requires two arguments
not(23)	because '23' is not of sort Bool (unless it's been declared, '23' doesn't have any sort!)
not() and true and false(not)	lots of reasons why this isn't well-formed!

Example 4.1 *More importantly, here are some examples of terms that are well-formed.*

1. true

2. true and false
3. not(true and false)
4. not(true and false) and not(true)
5. not(not(true and false) and not(true)) and true
6. not((not(not(true and false) and not(true)) and true) and not(false))

How do we know these are well-formed terms? If we look at Example 4.1(1), we note that true doesn't take any arguments, so just writing 'true' gives us a Bool, and so true is a term of sort Bool. Similarly, we can see that false, since it requires no arguments, is itself a term of sort Bool; since we already know that true is a term of sort Bool and we know that and is an infix argument that takes two Bools and returns a Bool, this explains why Example 4.1(2) is a well-formed term of sort Bool.

In a sense, terms are like problems, just like '2×6' is a problem whose answer is '12'. The term 'true and false' can be simplified, according to the truth table on page 31, to 'false'. We said before that writing terms was one of the really useful things that Maude lets us do; but if terms are problems, the *really* useful thing that Maude does for us is to give us the answer to those problems. This great boon doesn't come for free, however — in order to get solutions to problems, we have to write *equations*.

Exercise 4.1 *Before we look at equations, have one last look at Example 4.1. What are the answers to each of the problems that those well-formed terms represents?*

Equations

We've been looking at one Maude specification, BOOLEANS, and talked about names, sorts, operations, and terms. You may have forgotten by now that the specification had four equations in it, so let's remind ourselves of what they were. Here's the specification once again:

```
fmod BOOLEANS is

  sort Bool .

  ops true false : -> Bool .
  op not : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .

  eq not(true) = false .
  eq not(false) = true .

  var Y : Bool .

  eq true and Y = Y .
  eq false and Y = false .

endfm
```

The first equation says that if we apply not to true, the result is false. If we think of terms as problems, this equation gives us the answer to the problem not(true). The second equation gives the answer to the problem not(false). Taken together, these two equations give exactly the same information as the truth table for not:

X	$notX$
true	*false*
false	*true*

Note that this is an example of definition by exhaustion: there are two possible arguments to not; namely true and false: we have one equation saying what the output should be in each of these two cases.

The third and fourth equations are more cunning. We could have written

```
eq   true and true  =  true .
eq   true and false  =  false .
eq   false and true  =  false .
eq   false and false  =  false .
```

which would have exactly the same effect: both sets of equations would capture exactly the same and as we specified in the truth table on page 31. That is, they would give exactly the same answers to any problems (terms) that used 'and'. Specifically, the two equations

```
eq   false and true  =  false .
eq   false and false  =  false .
```

say that if the first argument is false, then it doesn't matter what the second argument is, the result is going to be false regardless, and that is exactly what is said by our equation

```
var  Y : Bool .
eq   false and Y  =  false .
```

Note how Y stands for both true and false. Similarly, the two equations

```
eq   true and true  =  true .
eq   true and false  =  false .
```

say that if the first argument is true, then the result will be the same as the second argument, whether that second argument is true or false, and that is exactly the meaning of the equation

```
var  Y : Bool .
eq   true and Y  =  Y .
```

Note that the two equations

```
eq   true and Y  =  Y .
eq   false and Y  =  false .
```

define and by induction on the first argument.

Summary

Maude is a notation for specifying abstract data types. Maude specifications come in modules: each module has a name, and consists of a number of declarations

4.4 Using Maude

Maude is a specification language, so its main function is to provide a notation in which specifications of ADTs can be written. But this would make it no better than UML.

The real power of Maude is in its use of equations to simplify terms. For example, consider the terms in Example 4.1. One of those terms was not(true and false). From the equation

```
var  Y : Bool .
eq   true and Y = Y .
```

we can see that this is equal to not(false). In turn, the equation

```
eq   not(false) = true .
```

tells us that this is equal to true. In fact, the equations in BOOLEANS are such that we can take any well-formed term of sort Bool (such as those in Example 4.1) and simplify it to either true or false.

So how do we get Maude to do this for us? The keyword reduce is followed by a well-formed term; Maude will simplify (reduce) the term using the equations that are available to it. Generally, the available equations are those in the last module that Maude has read — what do we meam by 'read'? Maude is an interpreted language, which means that Maude is actually a program that reads input from the user, and responds to that input. The input from the user can be modules or commands such as reduce-commands When you start Maude (usually, by typing 'maude' at the command-line), you'll see something like

```
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
           Maude 2.3 built: Feb 14 2007 17:53:50
           Copyright 1997-2007 SRI International
                  Fri Jun  4 01:21:22 2010
Maude>
```

When you see `Maude>`, this is the Maude *prompt*, which invites the user to type a Maude element: a module or a command. You *could* type in the module BOOLEANS at the prompt, but that would be tedious and error-prone; a more efficient way of working would be to use your favourite text-editor to create a file called 'booleans.maude', with the following text[3]:

```
set include BOOL off .

fmod  BOOLEANS  is

  sort  Bool .

  ops  true false : -> Bool .
  op  not : Bool -> Bool .
  op  _and_ : Bool Bool -> Bool .

  eq  not(true)  =  false .
  eq  not(false) =  true .

  var  Y : Bool .

  eq  true and Y  =  Y .
  eq  false and Y =  false .

endfm
```

Now start Maude, and at the Maude prompt, type 'in booleans'. This will make the Maude interpreter read in the file booleans.maude, and you should see something like

[3]The command set include BOOL off stops the interpreter looking at Maude's 'built-in' specification of the Booleans, which uses the same notation — if the interpreter sees two copies of the Booleans, it can get a bit confused.

```
Maude> in booleans
==========================================
fmod BOOLEANS
Maude>
```

The good thing here is that we see the Maude prompt. If there are syntax errors, Maude will — probably — let us know what they are. For example, if we misspell the operation name not when we create the file booleans.maude, we might see the following.

```
Maude> in booleans
==========================================
fmod BOOLEANS
Warning: "booleans.maude", line 36 (fmod BOOLEANS): bad token noy.
Warning: "booleans.maude", line 36 (fmod BOOLEANS): no parse for statement
eq noy (true) = false .
Maude>
```

means that there was an error on line 36 of the file booleans.maude. The message '`bad token noy`' means that the word 'noy' wasn't recognised (it should have been not).

If you do try typing in a module line-by-line at the Maude prompt, you'll see that Maude has two different prompts:

```
Maude> fmod BOOLEANS is
> sort Bool .
>
```

Every time you hit Return, you see the short prompt '`>`', which indicates that Maude is still waiting for you complete the module. Once you enter endfm (and hit return), you'll get back to the proper Maude prompt, `Maude>`. If you read in a file and see something like this:

```
Maude> in booleans
>
```

then there's some problem: Maude is still waiting for the input to be completed. Perhaps you forgot a full-stop, or maybe you didn't close a bracket; in any case, you'll have to go through your file looking for a syntax error. Type Control-C to get back to the proper Maude prompt.

Once you've fixed any errors, you can make Maude do some work. At the prompt, type reduce true and false . You should see this:

```
Maude> reduce true and false .
reduce in BOOLEANS : true and false .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
Maude>
```

Maude tells you which module she's working with (in this case, BOOLEANS), how many equations have been used (how many 'rewrites'), how much time it took, and, most importantly, what the result is: in this case, false.

You can put your reductions in a file. For example, you might create a file called boolTest.maude containing the following:

```
load booleans

red true .

red true and false .
```

(load is a non-verbose version of in; it tells Maude to read in a file, but not print out the names of the modules in the file). At the Maude prompt you can type in boolTest and you should see:

```
Maude> in boolTest
==========================================

==========================================
reduce in BOOLEANS : true .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in BOOLEANS : true and false .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false
Maude>
```

Exercise 4.2 *Use Maude to check your answers to Exercise 4.1.*

4.5 More Examples

4.5.1 Booleans, Again

In our specification of the booleans, we had only the constants true and false, negation (not), and conjunction (and). If we want to add an operation for disjunction, then we could edit our file booleans.maude and add a declaration and an equation for this operation, or we could write the following.

```
fmod BOOLEANS+OR is
  protecting BOOLEANS .
  op   or : Bool Bool → Bool .
  vars A B : Bool .
  eq   A or B = not(not(A) and not(B)) .
endfm
```

The keyword **protecting** is followed by the name of a module (and a space and a full-stop). The effect of this is to import the named module (BOOLEANS in this case), so that all the sorts, operations, and equations declared in that module can be used in the current module. Since BOOLEANS declares the sort Bool and the four operations true, false, not and and, they can all be used in BOOLEAN-OPERATIONS. This allows us to declare our three new operations (they all take Bools as arguments and return a Bool as result). It also allows us to write the equation that defines the new operation or using the operations not and and.

Exercise 4.3 *Write out truth tables for the terms A or B and not(not(A) and not(B)) to check that the equation defining or is correct.*

```
fmod BOOLEAN-OPERATIONS is
  protecting BOOLEANS .
  ops  _or_  _implies_  _xor_ : Bool Bool → Bool .
  vars A B : Bool .
  eq   A or B = not(not(A) and not(B)) .
  eq   A implies B = ... .
```

```
    eq   A xor B = ... .
endfm
```

Exercise 4.4 *Complete the specification BOOLEAN-OPERATIONS by filling in the right-hand sides of the last two equations to define the operations* implies *and* xor*. Solutions are given in the next specification, so do this before you read on. The truth table for exclusive-or is*

A	B	A *xor* B
true	*true*	*false*
true	*false*	*true*
false	*true*	*true*
false	*false*	*false*

Importing a module has the effect of including all the declarations of the imported module in the module that does the importing. So, for example, the module BOOLEAN-OPERATIONS as written above is equivalent to the following.

```
fmod BOOLEAN-OPERATIONS is

  sort Bool .
  ops true false : -> Bool .
  op not : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .

  eq not(true) = false .
  eq not(false) = true .

  var Y : Bool .
  eq true and Y = Y .
  eq false and Y = false .

  ops _or_ _implies_ _xor_ : Bool Bool -> Bool .

  vars A B : Bool .
  eq A or B = not(not(A) and not(B)) .
  eq A implies B = not(A) or B .
  eq A xor B = (not(A) and B) or (A and not(B)) .
endfm
```

There are usually many different but equivalent ways of writing specifications. We've already seen two different sets of equations that define and, and the operation xor could be defined as in the last equation above, or it could be defined by

```
  eq true xor B = not(B) .
  eq false xor B = B .
```

or even by

```
  eq true xor true = false .
  eq true xor false = true .
  eq false xor true = true .
  eq false xor false = false .
```

and perhaps you answered Exercise 4.4 in yet another way. So far, we've taken true, false, not and and as our 'building blocks' for the booleans, but we could also have taken true, false, not and or as the building blocks (exercise: define and using not and or). So with negation and one of and or or, it's possible to define all the other operations we might want; but can we make do with fewer operations? In 1913, Henry Sheffer showed that we only need true and false, and *one* binary operation (called the 'Sheffer stroke'):

```
fmod SHEFFER is
  sort Bool .
  ops true false : → Bool .
  op _|_ : Bool Bool → Bool .
  var Y : Bool .
  eq false | Y = true .
  eq true | true = false .
  eq true | false = true .
endfm
```

Exercise 4.5 *Using the equations in SHEFFER, write out a truth table for A | B.*

You should be able to see that A | B is equivalent to not(A and B), and if you know your logic gates, you'll recognise this as NAND. Now write Maude equations that define not, and and or, using only the Sheffer stroke.

4.5.2 Unary Numerals

The natural numbers are the numbers 0, 1, 2, 3, ... etc. You're familiar with decimal notation, where 110 means one hundred and ten, and you're probably also familiar with binary notation, where 110 means six. But we can also write the natural numbers in *unary* notation. This is sometimes also called 'caveman' notation, as unary notation can be thought of as scratches on a cave wall. The number six in unary notation would look like: |||||| . The number zero looks like:

(i.e., nothing: just a blank cave wall). We'll use Maude to specify the ADT of natural numbers, but since it's Maude and not a cave wall, we'll change the notation slightly; we'll write 0 for the blank cave wall, and we'll let s represent a scratch, so that 1 is written s(0), 2 is written s(s(0)), 3 is written s(s(s(0))), and so on. (You could also think of s as standing for 'successor', i.e., the next number.) Here's the Maude spec.

```
fmod NATURALS is

  sort Nat .

  op 0 : -> Nat [ ctor ] .
  op s : Nat -> Nat [ ctor ] .

endfm
```

You'll have noticed we've put ctor in square brackets at the end of the operation declarations. This stands for 'constructor', and indicates that these operations are only there to let us write terms. These are numerals, after all. A term such as s(s(s(s(s(s(0)))))) is nothing more than a numeral, a way of writing down the number six in unary notation. Another way of thinking of this is that these operations have no functionality: they don't *do* anything, and if we ask Maude to

```
reduce s(s(s(s(s(s(0)))))) .
```

then the result should just be s(s(s(s(s(s(0)))))), because that is just the way of writing down, in unary notation, the number six. On the other hand, an operation for addition (see below!) *would* have some functionality: if we had an addition operation and we asked Maude to

```
reduce s(s(0)) + s(0) .
```

then we would expect the answer to be s(s(s(0))), just as, in decimal notation, we would expect the answer to 2 + 1 to be given by a numeral, in this case: 3.

Maude doesn't treat constructors any differently to other operations; adding ctor to a declaration is just a way of telling the reader of the spec that an operation is no more than a 'building block' for constructing terms. (Well, there is a debugging facility in Maude that will check that the results of reductions are made up entirely of constructors. For example, if we asked Maude to reduce s(s(0)) + s(0), and the result was the self-same term s(s(0)) + s(0), then we'd suspect that we hadn't specified addition correctly, that our equations hadn't quite captured the functionality of addition. But we won't be concerned with this debugging facility — we'll just make sure that we get things right!)

Exercise 4.6 *Which operations in BOOLEANS should be declared as constructors? (The answer is given below, so answer this now, before you read on.)*

Well, we now have a specification of unary numerals, with no functionality at all, and we can write any natural number as a unary numeral . . . but does this have any use?

Not really.

So let's start adding some functionality. Let's begin by specifying a test for being equal to zero:

```
fmod TEST-ZERO is

  protecting NATURALS .
  protecting BOOLEANS .

  op isZero : Nat -> Bool .

  var N : Nat .
```

```
eq   isZero(0) = true .
eq   isZero(s(N)) = false .
endfm
```

The first equation here says that 0 is equal to zero. The second equation says that for any number N (whether N is 0 or s(0) or s(s(0)) or whatever) the number s(N) is *not* equal to zero.

All very sensible, but what's really interesting is how these two equations use the two constructors in NATURALS. Recall that there were two constructors, 0 and s, and then note that each of our two equations uses one of these constructors in the argument to isZero. Because NATURALS has two constructors, it follows that every unary numeral can be written using only these constructors. In other words, every unary numeral is *either* 0, or it has the form s(N) for some numeral N. In TEST-ZERO, we have one equation for each of these two possibilities.

We saw exactly the same thing in the definition of not in BOOLEANS. If you answered Exercise 4.6 correctly, you'll have said that Bool has two constructors: true and false. That means that — no surprises here — every Boolean is either true or false. The equations defining not take these two constructors as arguments to not:

```
eq   not(true) = false .
eq   not(false) = true .
```

and say, in each case, what the result should be.

For Booleans, both constructors are constants; for the naturals, one constructor, 0, is a constant, and the other, s, isn't: it takes a Nat as argument. Hence the form of the second equation, which uses a variable, N, as argument to s. Let's see if we can use this same trick to define addition.

Addition takes two arguments (the two numbers to be added together), and the result is a number. Since numbers are represented by numerals (unary numerals for us in this section), this *type* information is recorded by declaring the addition operation as follows.

```
op   _+_ : Nat Nat → Nat .
```

Now we want to go beyond type information and say *what* the output should be for any given inputs. Addition takes two inputs, but let's just concentrate on the first input; if it's 0, then we're adding nothing to the second input: the result should be whatever the second input is.

Let's call the second input N; we want to say that 0 + N is just N. And that's exactly what the following equation says.

```
var  N : Nat .
eq   0 + N = N .
```

But what if the first argument isn't 0? Well, in that case, the first argument must be of the form s(M) for some number M. This gives us the left-hand side of an equation:

```
vars M N : Nat .
eq   s(M) + N = ... .
```

Before we worry about the right-hand side, note that these two equations will apply to the addition of any two unary numerals. The variable N will match whatever number the second argument may be; and the first equation applies when the first argument is 0, and the second equation applies when the first argument isn't 0. Okay, on to the right-hand side.

Instead of thinking about scratches on a cave wall, let's think about pebbles (exercise: look up the etymology of 'calculate'). Imagine you have some pebbles in your left hand and some pebbles in your right hand, and you want to know how many you have altogether. You might count all the pebbles in your right hand (the second argument) and then count all the pebbles in your left hand (the first argument) one by one into your right hand. You stop when your left hand is empty (that's our first equation); if your left hand isn't empty, then you've got s(M) pebbles in there, so you pass one pebble on, leaving you with M pebbles in that hand (and leaving you with the simpler problem of adding M to s(N). So our second equation is

```
vars  M N : Nat .
eq    s(M) + N = M + s(N) .
```

Well, that's the theory; how does it work in practice? Let's pretend to be Maude. Suppose you type this at Maude's prompt:

```
reduce  s(s(0)) + s(s(0)) .
```

Maude looks at her list of equations to see which equations can be applied to this term. She only has two equations, and the first doesn't apply here, because the first argument to the addition operation isn't 0. The second does apply, and Maude notes that the variable M matches with the term s(0), and that the variable N matches with s(s(0)). Maude concludes that the whole term should therefore be equal to M + s(N); and since M is s(0) and N is s(s(0)), Maude says that

$$\mathsf{s(s(0))} + \mathsf{s(s(0))} \;=\; \mathsf{s(0)} + \mathsf{s(s(s(0)))}$$

and turns her attention to the term on the right. The first equation doesn't apply because the first argument isn't 0, but the second equation does apply, and this time M is 0 and N is s(s(s(0))), so the term s(0) + s(s(s(0))) simplifies to 0 + s(s(s(s(0)))), so we have

$$\mathsf{s(s(0))} + \mathsf{s(s(0))} \;=\; \mathsf{s(0)} + \mathsf{s(s(s(0)))} \;=\; \mathsf{0} + \mathsf{s(s(s(s(0))))}$$

and Maude turns her attention to the term on the right. This time, our second equation doesn't apply but the first equation does: adding zero to any number gives that number as result, so we have

$$\mathsf{s(s(0))} + \mathsf{s(s(0))} \;=\; \mathsf{s(0)} + \mathsf{s(s(s(0)))} \;=\; \mathsf{0} + \mathsf{s(s(s(s(0))))} \;=\; \mathsf{s(s(s(s(0))))} \;.$$

And at this point, Maude stops. Neither of her two equations applies to the term on the right (there is no +) so the result of the reduction is

$$\mathsf{s(s(s(s(0))))}$$

and Maude has proved that 2+2 = 4.

There's always more than one way of skinning cats or writing specifications. We can get the same functionality by replacing the second equation with this one:

```
vars  M N : Nat .
eq    s(M) + N = s(M + N) .
```

Using this equation, the problem of 2+2 is solved as follows:

$$\mathsf{s(s(0))} + \mathsf{s(s(0))} \;=\; \mathsf{s(\ s(0) + s(s(0))\ )} \;=\; \mathsf{s(s(\ 0 + s(s(0))\ ))} \;=\; \mathsf{s(s(s(s(0))))}$$

Exercise 4.7 *Define multiplication on unary numerals by completing the following equations.*

```
op   _*_ : Nat Nat → Nat .
vars M N : Nat .
eq   0 * N = ... .
eq   s(M) * N = ... .
```

Also, define exponentiation. Use Maude to check your answers.

4.5.3 Lists, the Queen of ADTs

There are lots of programming languages. Maude is a *declarative* language, which means that Maude 'programs' consist of declarations of sorts and operations, which are defined by equations. Java is an example of an *imperative* language, which means that Java programs consist of commands such as assignments. The first declarative language was LISP (for 'list processing'), which was based on Alonzo Church's λ-calculus, and had lists as a built-in data structure.

Here's a Maude specification of lists.

```
fmod LISTS is
  protecting INT .
  sort List .
  op [] : → List [ ctor ] .
  op _:_ : Int List → List [ ctor ] .
endfm
```

These two ctors let us write down any list of integers. The empty list is written [] (of course, we could have chosen any name we wanted, e.g., null or empty). We use an infix colon to add integers to a list, so for example, 1 : [] is a list with just one integer in it, and 3 : 2 : 1 : [] is a list with three integers. Because we only have these two constructors, any list is either empty, or is of the form I : L for some integer I and list L. We can use this fact in defining an operation to calculate how many numbers are in a list:

```
op   length : List → Int .
```

This operation takes a list as input; whatever list is given, it's either empty or of the form I : L. The case where the input list is empty is covered by this equation:

```
eq   length([]) = 0 .
```

The other case is covered by this equation:

```
var  I : Int .
var  L : List .
eq   length(I : L) = 1 + length(L) .
```

And between them, the two equations say how to calculate the length of any list. For example,

$$\begin{aligned} \mathsf{length}(3 : 2 : 1 : []) &= 1 + \mathsf{length}(2 : 1 : []) \\ &= 1 + 1 + \mathsf{length}(1 : []) \\ &= 1 + 1 + 1 + \mathsf{length}([]) \\ &= 1 + 1 + 1 + 0 \\ &= 3 \, . \end{aligned}$$

Exercise 4.8 *Specify an operation that adds up all the numbers in a list.*

Specify an operation that calculates the product of all the numbers in a list (i.e., multiply them all together)

Specify an operation that computes the average of the numbers in a list, rounded down to an integer value (i.e., use Maude's integer division operation, quo — which is an infix operation).

Use Maude to check your answers.

Exercise 4.9 *Specify an infix operation _++_ that takes two lists and concatenates them. (Hint: just like for addition, use induction on the first argument.)*

Specify an operation that reverses a list; for example, given a list 1 : 2 : 3 : [], this operation should return the list 3 : 2 : 1 : []. (Hint: use concatenation.)

Use Maude to check your answers.

Suppose we wanted to remove the last element of a list.

```
op  removeLast : List → List .
```

For example, we would want removeLast(3 : 2 : 1 : []) = 3 : 2 : []. We can't remove the last element from the empty list, so we have

```
eq  removeLast([]) = [] .
```

What about when the input list is of the form I : L? Well, if L is empty, then I is the last element, so the result should be the empty list:

```
eq  removeLast(I : []) = [] .
```

Now if L is not empty, then it is of the form J : L' for some integer J and list L', and so we can write

```
vars I J : Int .
var  L : List .
eq   removeLast(I : J : L) = I : removeLast(J : L) .
```

This will work (exercise: try out removeLast(3 : 2 : 1 : [])), but we can give a clearer definition using *conditional equations*.

A conditional equation is only applied if a given condition holds. For example, we might want to say that the equation

```
eq  removeLast(I : L) = I : removeLast(L) .
```

should only be applied if L is not empty. Conditional equations are written in Maude using the keyword **ceq** (or **cq**) and the keyword **if** in the format:

```
ceq  left-hand side = right-hand side  if  condition .
```

The *condition* is a term of sort Bool. Maude has a built-in module BOOL that declares the sort Bool (so our specification BOOLEANS was unnecessary!), and because Bool is used in conditional equations, we don't need to explicitly import BOOL; it's imported by default into every module — that was why we had to include the command

```
set include BOOL off .
```

in booleans.maude, to prevent Maude getting confused by two different versions of true and false.

Back to removeLast: we can specify the same functionality as above using conditional equations as follows.

```
fmod REMOVE-LAST is
  protecting LISTS .
  op removeLast : List → List .
  var I : Int .
  var L : List .
  eq removeLast([]) = [] .
  ceq removeLast(I : L) = L if L == [] .
  ceq removeLast(I : L) = I : removeLast(L) if L =/= [] .
endfm
```

The conditions here use Maude's built-in equality test ==, and inequality test =/=. (See the lecture notes for how Maude uses conditional equations: basically, they're only applied if the condition evaluates to true.)

Exercise 4.10 *Specify an operation isIn that takes an integer and a list, and returns true if the integer occurs in the list and false otherwise. For example, isIn(2, 1 : 2 : 3 : []) should be true, and isIn(4, 1 : 2 : 3 : []) should be false.*

4.5.4 Binary Numerals

```
fmod BINARY-DIGITS is
  sort BinaryDigit .
  ops 0 1 : → BinaryDigit .
endfm

fmod BINARY-NUMERALS is
  protecting BINARY-DIGITS .
  sort BinaryNumeral .
  subsort BinaryDigit < BinaryNumeral .
  op _ _ : BinaryNumeral BinaryDigit → BinaryNumeral .
endfm

fmod BINARY-NUMERALS-SEMANTICS is
  protecting BINARY-NUMERALS .
  protecting INT .
  op [[ _ ]]D : BinaryDigit → Int .
  var D : BinaryDigit .
  var N : BinaryNumeral .
endfm
```

4.5.5 Stacks, the Old King of ADTs

```
fmod STACKS is
  protecting INT .
```

```
  sort  Stack .

  op    empty  :  → Stack .
  op    push  : Int Stack → Stack .
  op    top  : Stack → Int .
  op    pop  : Stack → Stack .

  var   I : Int .
  var   S : Stack .
  eq    top(push(I, S)) = I .
  eq    pop(push(I, S)) = S .
endfm

fmod STACKS is

  protecting INT .

  sort  Stack .

    *** non-empty stacks
  sort  NEStack .
  subsort NEStack < Stack .

  op    empty  :  → Stack .
  op    push  : Int Stack → NEStack .
  op    top  : NEStack → Int .
  op    pop  : NEStack → Stack .

  var   I : Int .
  var   S : Stack .
  eq    top(push(I, S)) = I .
  eq    pop(push(I, S)) = S .
endfm
```

Chapter 5

The Semantics of Maude

in which we learn the meaning of many things

It is a fact that Euclid's enunciations not infrequently leave something to be desired in point of clearness and precision.

Sir Thomas L. Heath (ed.)
footnote to Proposition 4 of Euclid's *Elements*

---

use Euclid to motivate the notion of model

Maude is a language for specifying abstract data types, so the semantics of Maude must have something to do with abstract data types. But you don't specify abstract data types just for the fun of it; the end goal is to get an implementation of a software system that uses those data types, so the semantics of Maude must also have something to do with possible implementations. So in order to say what a Maude specification means, we'll have to say something about both the abstract data type that is being specified, as well as the possible ways of implementing the specified data type.

In this chapter, we'll say exactly what ADT is specified by a Maude specification, and we'll also say what it means to implement a Maude specification.

In fact, since an abstract data type consists of abstract data values together with the operations that use those values, any software system that uses an ADT must itself be an ADT, so Maude is really a language for specifying software systems. Really, then, what this chapter is all about is how to specify and implement software systems.

We need some way of going from an arbitrary Maude specification to its meaning as an actual software system. The following section presents *signatures* as an abstract way of talking about the sorts and operations that are declared in a Maude specification.

5.1 Signatures: Maude and Syntax

The previous chapter had lots of different Maude specifications. In this chapter we want to talk about *all* Maude specifications: what do they mean? In order to talk about all Maude specifications without actually listing all the possible Maude specifications, we need some way of talking about an arbitrary specification.

Was there anything all the specifcations we looked at had in common? Yes: they were all Maude specifications. That means they all consisted of a named module that contained declarations of sorts, operations and equations. For the moment, we'll forget about equations, and concentrate on the sorts and operations.

Looking first at the sorts, a Maude specification declares some number of sorts: there were none in the module NOTHING; just one sort, Bool, in the module BOOLEANS; two sorts, BinaryDigit and BinaryNumeral, in the module BINARY-NUMERAL; and so on. So we can say that every Maude specification gives us a set

of sorts (or sort-*names*, to be precise, but to save ink, let's just say 'sort' instead of 'sort name' from now on).

We could also say that every Maude specification gives us a set of operations, but there's more to operations than just names: each operation has a *type*. The type of an operation is the number of inputs it requires, the sorts of those inputs, and the sort of its output — in short, its type is what follows the colon (**:**) in the declaration. In the algebraic-specification literature, it's more common to use the term *arity* than *type*. For example, the arity of

```
op  succ : Nat → Nat .
```

is 'one input of sort Nat and the result is of sort Nat'. That's quite a mouthful, so let's just separate the input and output sorts, and say that the arity of succ is (Nat, Nat) — the first element of the pair is the input sort, and the second element is the output sort. The second element (the output sort) is always going to be just a single sort; the first element will be a *list* of sorts. For example, the arity of

```
op  _+_ : Nat Nat → Nat .
```

is (Nat Nat, Nat), and the arity of

```
op  _ _ : BinaryNumeral BinaryDigit → BinaryNumeral .
```

is (BinaryNumeral BinaryDigit, BinaryNumeral).

What about constants, the operations that take no inputs? Computer scientists love to give names to nothing, so let's write '[]' for the particular kind of nothing that is the empty list of input sorts. And now the arity of

```
op  0 :  → Nat .
```

is ([], Nat).

Exercise 5.1 *What are the arities of*

- *op nil :* → *List*
- *op _ _ : Int List* → *List*

Back to our muttons: a Maude specification gives us a set of sorts; let's call this set S. For example, from the specification BINARY-NUMERALS, we have $S = \{\mathsf{BinaryDigit}, \mathsf{BinaryNumeral}\}$. Each operation has an arity, which is a pair (w, s), where $s \in S$ (i.e., s is a sort declared in the specification), and this is the output sort of the operation; and $w \in S^*$ (i.e., w is a list of sorts), and these are the sorts of the inputs to the operation. Given a Maude specification, let's write $O_{w,s}$ for the set of all of the operations of arity (w, s). For example, for the specification BOOLEANS, we have $S = \{\mathsf{Bool}\}$ and

- $O_{[\,],\,\mathsf{Bool}} = \{\mathsf{true}, \mathsf{false}\}$.
- $O_{\mathsf{Bool},\,\mathsf{Bool}} = \{\mathsf{not}\}$
- $O_{\mathsf{Bool\ Bool},\,\mathsf{Bool}} = \{\mathsf{and}\}$

Moreover,

- $O_{\mathsf{Bool\ Bool\ Bool},\,\mathsf{Bool}} = \{\}$, and
- $O_{\mathsf{Bool\ Bool\ Bool\ Bool},\,\mathsf{Bool}} = \{\}$,

and $O_{w,\,\mathsf{Bool}} = \{\}$ for any w other than $[\,]$, Bool, or Bool Bool.

Interlude. A pair consists of two things, and is usually written with round brackets. For example, $(1, 15)$ is a pair of integers, and $(7, \texttt{"hello"})$ is a pair whose first component is an integer (7), and whose second component is a string (`"hello"`). The *types* of these pairs are given by the types of their first and second components, separated by the symbol '$\times$': the type of $(1, 15)$ is $Int \times Int$ and the type of $(7, \texttt{"hello"})$ is $Int \times String$. What we've called a type is really just a set: *Int* is the set of all integers; *String* is the set of all strings.

Generally, if A and B are sets, we write $A \times B$ for the set of all pairs whose first component is an element of A and whose second component is an element of B. For example, $(true, 28) \in \{true, false\} \times Int$. If S is the set of sorts in a Maude specification, then an arity is an element of $S^* \times S$, because the first component of an arity is a list of sorts (the sorts of the inputs), and the second component is a sort (the result sort).

But why use the notation '$\times$'? If S is a set, the *cardinality* of S, usually written $\#S$, is the number of elements of S. So $\#\{true, false\} = 2$ and $\#\{0\} = 1$.

Exercise 5.2 *How many elements are there of* $\{0, 1, 2\} \times \{true, false\}$*?*

Exercise 5.3 *Sometimes you'll see people write* B^A *for the set of all functions* $A \to B$ *(input from the set A; output of sort B) — why do they use that notation?*

So now you can guess why the notation '$\emptyset$' is used for the empty set. Sometimes the notation '1' is used for the set $\{\emptyset\}$, and sometimes it's used for the set $\{0\}$. In one way, it doesn't matter which set we choose for '1', because all sets that contain only one element are just as good as each other: they all contain only one element, so those elements are in a one-to-one correspondence (that correspondence is, for example, $\emptyset$ - 0).

Exercise 5.4 *Whatever set you choose for '1' (*$\{\emptyset\}$*,* $\{0\}$*, or* $\{25\}$*), what is the cardinality of:*

1. $1 \times \{0, 1, 2\}$
2. $1 \times A$*, for any set A, and*
3. A^1*, for any set A?*

We can summarise all the sort- and operation-declarations in a Maude specification by giving the set S of sorts and all the sets $O_{w,\,s}$ of operations of arity (w, s) (as the example above suggests, most of the sets $O_{w,\,s}$ will be empty). This abstract way of looking at the sorts and operations declared in a Maude specification is what we call a *signature*.

Definition 5.1 *A* ***signature*** *is a pair* (S, O)*, where* S *is a set, and* O *is a family of sets* $O_{w,\,s}$ *for* $w \in S^*$ *and* $s \in S$*.*

Every Maude specification has a signature: we let S be the set of all the sorts that are declared in the specification; and each set $O_{w,\,s}$ consists of all the operations that are declared in the specification to have arity (w, s). Moreover, every signature gives rise to a Maude specification, so signatures and specifications are just the same thing (at least, as long as we're forgetting about equations).

Example 5.1 *The signature of the specification* NATURALS *is the pair* $(S^{\mathsf{N}}, O^{\mathsf{N}})$*, where* $S^{\mathsf{N}} = \{\mathsf{Nat}\}$ *and*

$$\begin{aligned} O^{\mathsf{N}}_{[\,],\,\mathit{Nat}} &= \{\mathsf{0}\} \\ O^{\mathsf{N}}_{\mathit{Nat},\,\mathit{Nat}} &= \{\mathsf{succ}\} \end{aligned}$$

and $O^{\mathsf{N}}_{w,\,s} = \emptyset$ *for all other* w*,* s*.*

Exercise 5.5 *What is the signature of* BINARY-DIGITS*?*

Note that we said every Maude *specification* has a signature; we didn't say every Maude *module*. So what's the difference between a Maude specification and a Maude module?

Well, a Maude module contains declarations of sorts and operations, but it can also import other modules, using the protecting keyword. For example, BOOLEAN-OPERATIONS imports BOOLEANS (see page 41). Importing another module is really just taking all the declarations in the imported module (and in the modules that that module imports, and so on), and including them in the importing module. When we talk about a *specification*, we mean the module that we would get if we literally included all the declarations from all the imported modules, and all the modules imported by those modules, and so on. For example, BOOLEAN-OPERATIONS is a module, but when we refer to BOOLEAN-OPERATIONS as a *specification*, what we mean is BOOLEAN-OPERATIONS-SPEC (page 42). The signature of BOOLEAN-OPERATIONS is, therefore, the signature of BOOLEAN-OPERATIONS-SPEC.

Exercise 5.6 *What is the signature of BOOLEAN-OPERATIONS (i.e., of BOOLEAN-OPERATIONS-SPEC)? What is the signature of BINARY-NUMERALS?*

Why do we need to bother with signatures at all? Why not just talk about Maude specifications (or even modules)? Well, we're going to be talking about what Maude specifications *mean*. For this, we need to talk about arbitrary specifications, and the declarations in those specifications. We will want to say things like: 'Suppose we have some Maude specification that declares a sort (let's call it s) and also declares a constant of sort s (let's call it c)...'. Now that we've introduced the notion of signature, we can say the same thing much more briefly as: 'Let (S, O) be a signature, and let $c \in O_{[\,],\,s}$...'. This is just the sort of thing we do in the next section.

Interlude. Algebra and naming: liber amicorum quote.

5.2 Models: Implementing Specifications

Why do we write Maude specifications?

The goal is always to write a program that does something useful or at least entertaining, and a program that does something useful or entertaining makes use of some data that is structured in one way or another. Maude allows us to say how data is structured, and what you can do with that data, so we can use Maude to specify useful and interesting programs. We should be able to write a Maude specification, then take that specification to a programmer, hand it over and say 'implement that for me, please'. The programmer will then go away for a few days (or a few months, or possibly even years) and then come back with a program. Obviously, we would hope that the program they give to us bore some relation to the specification we gave to them. The particular relationship we would hope for is crystallised in the notion of *model*: this is what we call a possible implementation of a Maude specification.

Let's forget about our hypothetical programmer and ask ourselves how *we* would go about implementing a Maude specification.

Each sort that's declared in a Maude specification gives a set of abstract data values that have to be given a concrete representation. For example, the BOOLEANS specification has a sort Bool with two abstract data values true and false — how will we implement these? Most programming languages actually do implement Booleans. Java implements Booleans as the data type `boolean`, with keywords `true` and `false`: these keywords are the concrete representation of the abstract data values. The language C implements Booleans in a different way, by using integers: false is represented by the integer `0`, and true is represented by the integer value `1` (actually, any non-zero integer value will be taken as true in C, but most C programmers will use `1`, rather than, say, `2014` — which would seem a bit arbitrary).

Once we've chosen a concrete representation for the abstract data values, we can turn our attention to the operations. For example,

op and : Bool Bool → Bool .

has to be implemented as something (a program, a method, or a bit of code) that will take two Booleans and return a Boolean — but we've already made a decision as to how Booleans are to be represented (let's pretend), so we have to work with those representations. This means that, if we've gone down the Java road and used keywords to represent the Booleans, then the inputs will each be either the keyword `true` or the keyword `false`. On the other hand, if we went down the C road and used the integers to represent the Booleans, then the inputs will be integers. If you're not familiar with C, it may seem odd to use 0 for false and any non-negative integer for true, but the designers of C weren't daft, and there are some very neat things that follow from the decision to use integers in that way. For example, and is implemented by multiplication: for any integers x and y, x and $y = xy$; the product xy is 0 (false) exactly when one of x and y is 0.

We can now say what an acceptable implementation is: for each sort, a set of concrete data representations; for each constant, an element from the set of representations; and for each operation, a function that takes inputs from the appropriate sets of representations and returns a result that is also in the appropriate set of representations. We call acceptable implementations *models*[1]

Definition 5.2 *Let $\Sigma = (S, O)$ be a signature. A Σ-**model** M consists of:*

1. *for each sort $s \in S$, a set M_s*
   *(the set of concrete representations of the abstract data values of sort s, we call these sets the **carrier sets**);*

2. *for each constant $c \in O_{[\,],\,s}$, an element $M_c \in M_s$; and*

3. *for each operation $o \in O_{w,\,s}$, where $w = s_1\, s_2\, \ldots\, s_n$, a function*
   $M_o \;:\; M_{s_1} \times M_{s_2} \times \cdots \times M_{s_n} \;\rightarrow\; M_s$.

That's quite complicated, so let's go straight to an example. We'll describe a model, R (for Reasonable) of BINARY-NUMERALS. Exercise 5.6 asks you to describe the signature (let's call it $\Sigma^{\mathsf{B}} = (S^{\mathsf{B}}, O^{\mathsf{B}})$) of this specification. Since the specification declares two sorts, BinaryDigit and BinaryNumeral, we have $S^{\mathsf{B}} = \{\mathsf{BinaryDigit}, \mathsf{BinaryNumeral}\}$. According to Definition 5.2(1), our model R must therefore have two carrier sets $R_{\mathsf{BinaryDigit}}$ and $R_{\mathsf{BinaryNumeral}}$. Let these sets be defined as

$$R_{\mathsf{BinaryDigit}} \;=\; \{0, 1\} \tag{5.1}$$

$$R_{\mathsf{BinaryNumeral}} \;=\; \{0, 1\}^* \;. \tag{5.2}$$

I.e., $R_{\mathsf{BinaryDigit}}$ is the set containing the two values 0 and 1 and $R_{\mathsf{BinaryNumeral}}$ is the set of lists of 0s and 1s.

There are two constants of sort BinaryDigit declared in BINARY-NUMERALS, so we have $O_{[\,],\,\mathsf{BinaryDigit}} = \{\mathsf{0}, \mathsf{1}\}$. According to Definition 5.2(2), Each of these constants should be implemented as elements R_{0} and R_{1}; both of these should belong to the carrier set $R_{\mathsf{BinaryDigit}}$. We choose (unsurprisingly)

$$R_{\mathsf{0}} \;=\; 0 \tag{5.3}$$

$$R_{\mathsf{1}} \;=\; 1 \;. \tag{5.4}$$

There's just one more operation declared in BINARY-NUMERALS:

op _ _ : BinaryNumeral BinaryDigit → BinaryNumeral .

[1] Sometimes, we call these *algebras*; both terms — *model* and *algebra* — are well-established, as is the random and often inconsistent choice of one or the other. Sorry.

According to Definition 5.2(3), which talks about '$o \in O_{w,\,s}$, where $w = s_1\, s_2\, \ldots\, s_n$', and noting that

$$\_\ \_ \in O^{\mathsf{B}}_{\mathsf{BinaryNumeral\ BinaryDigit,\ BinaryNumeral}}\ ,$$

we see that, in this case, $n = 2$, $s_1 = \mathsf{BinaryNumeral}$, $s_2 = \mathsf{BinaryDigit}$, and $s = \mathsf{BinaryNumeral}$, all of which means that $R\_\ \_$ has to be implemented as a function of type $R_{\mathsf{BinaryNumeral}} \times R_{\mathsf{BinaryDigit}} \to R_{\mathsf{BinaryNumeral}}$. So $R\_\ \_$ takes two inputs: the first will be a list of 0s and 1s; and the second input will be a 0 or a 1. To define $R\_\ \_$, we say: for any $N \in \{0,1\}^*$ and $D \in \{0,1\}$

$$R\_\ \_\ (N, D) = ND\ .$$

(On the right-hand side of this last equation, ND is the list comprising everything in the list N, followed by D, cf. the notation used in LIST.)

Here's another example of a model of the signature of BINARY-NUMERALS, which we'll call Q, for Quite Reasonable. This model interprets numerals as numbers, so the carrier-sets are numbers:

$$\begin{aligned} Q_{\mathsf{BinaryDigit}} &= \{0, 1, 2, \ldots\} \\ Q_{\mathsf{BinaryNumeral}} &= \{0, 1, 2, \ldots\} \end{aligned}$$

The constants and operations are implemented in a way very similar to the semantics of binary numerals

$$\begin{aligned} Q_{\mathsf{0}} &= 0 \\ Q_{\mathsf{1}} &= 1 \\ Q\_\ \_\ (N, D) &= 2 \times N + D \end{aligned}$$

Definition 5.2 says that $Q\_\ \_$ must have type $Q_{\mathsf{BinaryNumeral}} \times Q_{\mathsf{BinaryDigit}} \to Q_{\mathsf{BinaryNumeral}}$, so in the last equation above, both N and D are numbers: $N, D \in \{0, 1, 2, \ldots\}$.

Here's one more example of a model of the signature of BINARY-NUMERALS, which we'll call S, for Slightly Less Reasonable.

$$\begin{aligned} S_{\mathsf{BinaryDigit}} &= \{\mathit{true}, \mathit{false}\} && (5.5) \\ S_{\mathsf{BinaryNumeral}} &= \{\mathit{true}, \mathit{false}\} && (5.6) \\ S_{\mathsf{0}} &= \mathit{true} && (5.7) \\ S_{\mathsf{1}} &= \mathit{false} && (5.8) \\ S\_\ \_\ (N, D) &= N \wedge D && (5.9) \end{aligned}$$

Exercise 5.7 *Check that S really is a Σ^{B}-model, according to Definition 5.2.*

Define your own model T (for Totally Unreasonable) of the signature of BINARY-NUMERALS. Bonus points for making it totally unreasonable, but still conforming to Definition 5.2!

Here's another not-at-all-reasonable example of a model. This time, though, we'll define the not-at-all-reasonable model for *any* signature.

Definition 5.3 *Let $\Sigma = (S, O)$ be a signature. The 'Lazy Programmer' model, L_Σ, is defined as follows. First, the carrier sets: for each $s \in S$,*

$$(L_\Sigma)_s = \{0\}\ .$$

Yes, that's right: the lazy programmer uses only 0 as a concrete representation for ... well, everything. That gives us a clue as to how the constants are going to be implemented: for every sort $s \in S$, and every constant $c \in O_{[\,],\,s}$ of sort s,

$$(L_\Sigma)_c = 0\ .$$

(This is, of course, the only possible way that each constant can be implemented.) It's the same story for the other operations. If we have $o \in O_{w,\,s}$, where $w = s_1 \ldots s_n$, then o has to implemented as a function $(L_\Sigma)_o : (L_\Sigma)_{s_1} \times \cdots \times (L_\Sigma)_{s_n} \to (L_\Sigma)_s$. Each input must be 0, and the output must also be 0, so the only possible way this function can be defined is $(L_\Sigma)_o(0, \ldots, 0) = 0$.

This is a general recipe for making a model for any possible signature (i.e., for any possible Maude specification without equations). If we're interested in models as possible implementations of specifications, it might seem rather disappointing. We can make it a bit more interesting by noting that it proves

Proposition 5.1 *Every signature has at least one model.*

But we can do better than this; we can show that every signature has an *interesting* model that's built from terms. First, we define what terms are; the following is a more mathematical version of the discussion of terms in Section 4.3.

Definition 5.4 *Let $\Sigma = (S, O)$ be a signature. For each sort $s \in S$, the set $(T_\Sigma)_s$ of terms of sort s is defined inductively by:*

1. *for each constant $c \in O_{[],\,s}$,*
$$c \in (T_\Sigma)_s$$
i.e., every constant symbol is also a term.

2. *for each operation $o \in O_{s_1 \ldots s_n,\,s}$, and for all terms $t_i \in (T_\Sigma)_{s_i}$ $(i = 1..n)$,*
$$o(t_1, \ldots, t_n) \in (T_\Sigma)_s$$
i.e., writing o (as a symbol), followed by '(', followed by all the terms t_i separated by commas, and followed by ')', gives a term of sort s.

Remember we said that any $c \in O_{[],\,s}$ should really be called a constant *symbol*, and any $o \in O_{w,\,s}$ should be called an operation *name*? Definition 5.4 relies upon these things being symbols and names: they are syntactic things that can be written down. Definition 5.4(1) says that any constant symbol $c \in O_{[],\,s}$ is just a symbol: writing it down, and thinking of it as a string, is one way of writing a *term*. As an example of this definition, let $\Sigma^{\mathsf{L}} = (S^{\mathsf{L}}, O^{\mathsf{L}})$ be the signature of LISP lists:

```
fmod LISP-LISTS is
  protecting NATURALS .
  sort List .
  op nil : → List .
  op cons : Nat List → List .
endfm
```

Since $\mathsf{0} \in O^{\mathsf{L}}_{[],\,\mathsf{Nat}}$ and $\mathsf{nil} \in O^{\mathsf{L}}_{[],\,\mathsf{List}}$, Definition 5.4(1) tells us that

$$\begin{aligned} \mathsf{0} &\in (T_{\Sigma^{\mathsf{L}}})_{\mathsf{Nat}} && \text{i.e., 0 is a term of sort Nat, and} \\ \mathsf{nil} &\in (T_{\Sigma^{\mathsf{L}}})_{\mathsf{List}} && \text{i.e., nil is a term of sort List.} \end{aligned}$$

And since $\mathsf{succ} \in O^{\mathsf{L}}_{\mathsf{Nat},\,\mathsf{Nat}}$, Definition 5.4(2) tells us that (with $n = 1$, and $s_1 = s = \mathsf{Nat}$, and $t_1 = \mathsf{0}$)

$$\mathsf{succ(0)} \in (T_{\Sigma^{\mathsf{L}}})_{\mathsf{Nat}} \ .$$

Similarly, since $\mathsf{cons} \in O^{\mathsf{L}}_{\mathsf{Nat\ List},\,\mathsf{List}}$, Definition 5.4(2) tells us that (with $n = 2$, and $s_1 = \mathsf{Nat}$, $s_2 = \mathsf{List}$, $s = \mathsf{List}$, and $t_1 = \mathsf{0}$ and $t_2 = \mathsf{nil}$)

$$\mathsf{cons(0,\ nil)} \in (T_{\Sigma^{\mathsf{L}}})_{\mathsf{List}} \ .$$

Exercise 5.8 *Say why, according to Definition 5.4, cons(succ(0), cons(0, nil)) is a term of sort List.*

Interlude.

Exercise 5.9 *Why would it be reasonable to use the notation '1' for the Lazy-programmer model?*

Exercise 5.10 *Definition 5.2 makes a distinction between constants ($c \in O_{[\,],\,s}$) and other operations ($o \in O_{w,\,s}$, where $w = s_1 \ldots s_n$). We don't need to make that distinction. We could define, for any $w \in S^*$, the set M_w by*

$$M_w = \begin{cases} 1 & \text{if } w = [\,] \\ M_{s_1} \times \cdots \times M_{s_n} & \text{if } w = s_1 \ldots s_n \end{cases} .$$

Now we can replace Definition 5.2(2) and (3) with:

- *for every operation $o \in O_{w,\,s}$, a function $M_o : M_w \to M_s$.*

Why does this work; i.e., why does this cover Definition 5.2(2)? (Hint: see Exercise 5.4.)

Definition 5.4 just defines the sets $(T_\Sigma)_s$ of terms of sort s. These will be the carrier sets of a model (the *term algebra*[2]), but before we can say that this really is a model, we need to say how the constants and operations are implemented.

Definition 5.5 *Let $\Sigma = (S, O)$ be a signature. The* ***Σ-term algebra****, T_Σ has:*

1. *carrier sets $(T_\Sigma)_s$, for $s \in S$;*
2. *for $c \in O_{[\,],\,s}$, we define $(T_\Sigma)_c = c$; and*
3. *for $o \in O_{w,\,s}$, where $s = s_1 \ldots s_n$, we define*

$$(T_\Sigma)_o \ : \ (T_\Sigma)_{s_1} \times \cdots \times (T_\Sigma)_{s_n} \to (T_\Sigma)_s$$

by: for all $t_i \in (T_\Sigma)_{s_i}$ $(i = 1..n)$,

$$(T_\Sigma)_o(t_1, \ldots, t_n) = o(t_1, \ldots, t_n) \ .$$

Okay, there are some weird things going on in this definition. We blithely wrote '$(T_\Sigma)_c = c$', but 'c' is used in two different ways here. On the left-hand side, c is a constant symbol: $c \in O_{[\,],\,s}$, and Definition 5.2(2) tells us that $(T_\Sigma)_c$ has to be an element of the appropriate carrier set — in this case, an element of $(T_\Sigma)_s$. On the right-hand side, c is to be understood as an actual element of $(T_\Sigma)_s$, a term of sort s. But this is exactly what Definition 5.4(1) tells us: every constant is a term.

Similarly, we blithely wrote

$$(T_\Sigma)_o(t_1, \ldots, t_n) = o(t_1, \ldots, t_n)$$

and here '$(t_1, \ldots, t_n)$' is used in two different ways. On the left-hand side, it's the arguments to the function $(T_\Sigma)_o$; on the right-hand side, it's a string. Once again, it's Definition 5.4 that allows us to make sense of this. The right-hand side, $o(t_1, \ldots, t_n)$, is to be read as a sequence of symbols: o (as a symbol), followed by '(', followed by all the terms t_i separated by commas, and followed by ')', which is a term of sort s. In summary, the term algebra T_Σ implements every operation $o \in O_{w,\,s}$ as a term-building function that takes arguments t_1, ..., t_n and returns the term '$o(t_1, \ldots, t_n)$'.

Interlude. We've given two different ways of constructing a model for any signature: the Lazy-programmer model and the term algebra. Are they different? They certainly look different, but what about this specification?

```
fmod ONE is
  sort One .
  op 0 : → One .
endfm
```

[2] Told you we'd mix up the terms *model* and *algebra*

What's the term algebra for this spec?

We have shown that T_Σ is a Σ-model of any signature Σ, which gives us another proof of Proposition 5.1. The term algebra T_Σ is a much more interesting model than the Lazy-programmer model because terms are problems that are solved by models (and the lazy programmer solves all problems in a trivial way). This is what makes the term algebra really interesting: every specification just introduces some names (for sorts, constants and operations), and every model of a specification *interprets* these names — sort-names are interpreted as sets, constant-names are interpreted as elements of the appropriate carrier set, and operation-names are interpreted as functions taking inputs of the appropriate sorts (the carrier sets of the model) and returning an outputof the appropriate sort (the carrier set of the model). If we think of a model as an interpretation, this interpretation extends to terms.

For example, let's look at terms over the signature of NATURALS. Recall that this is a particularly simple signature: just one sort, Nat; one constant, 0; and one unary operator, succ. Here are two models of this signature:

$A_{\mathsf{Nat}} = \{0, 1, 2, \ldots\}$	$B_{\mathsf{Nat}} = \{true, false\}$
$A_{\mathsf{0}} = 0$	$B_{\mathsf{0}} = true$
$A_{\mathsf{succ}}(x) = x + 1$	$B_{\mathsf{succ}}(x) = \neg\, x$

Model A implements the constant 0 as the number 0, so when we view 0 as a term, its interpretation in A is A_{0}, i.e., 0. The interpretation of the term 0 in B is B_{0}, i.e., *true*.

Model A implements succ as the function that adds 1 to its argument. The interpretation of the term succ(0) is $A_{\mathsf{succ}}(A_{\mathsf{0}})$, i.e., $0 + 1$, which is, of course, just 1.

Model B implements succ as logical negation, so the term succ(0) is interpreted as $B_{\mathsf{succ}}(B_{\mathsf{0}})$, i.e., $\neg\, B_{\mathsf{0}} = \neg\, true = false$.

Similarly, the term succ(succ(0)) will be interpreted:

- in A as $A_{\mathsf{succ}}(A_{\mathsf{succ}}(A_{\mathsf{0}})) = 0 + 1 + 1 = 2$; and
- in B as $B_{\mathsf{succ}}(B_{\mathsf{succ}}(B_{\mathsf{0}})) = \neg\neg\, true = true$.

In general, a term succ(... succ(0) ...) which has, say, n succs will be interpreted in A as $A_{\mathsf{succ}}(\ldots A_{\mathsf{succ}}(A_{\mathsf{0}}) \ldots)$ — which will be n, so model A gives the obvious semantics to unary numerals. The same term will be interpreted in B as $B_{\mathsf{succ}}(\ldots B_{\mathsf{succ}}(B_{\mathsf{0}}) \ldots) = \neg \cdots \neg\, true$, with n negations — so B tells us whether a unary numeral is even or odd.

Even more generally, given any signature Σ, and any Σ-model M, any Σ-term is interpreted by replacing each constant c with M_c, and each operation o with M_o. In fact, this gives us a function h mapping terms to elements of the model M; to be precise, we have a family of functions $h_s : (T_\Sigma)_s \to M_s$ for each sort s, mapping terms of sort s to elements of the carrier set M_s.

Definition 5.6 *For signature $\Sigma = (S, O)$ and Σ-model M, the functions $h_s : (T_\Sigma)_s \to M_s$ for each sort s are defined by:*

1. *for each constant $c \in O_{[\,],\,s}$, let $h_s(c) = M_c$; and*
2. *for each operation $o \in O_{s_1 \cdots s_n,\,s}$ and $t_i \in (T_\Sigma)_{s_i}$ $(i = 1..n)$, let*

$$h_s(o(t_1, \ldots, t_n) = M_o(h_{s_1}(t_1), \ldots, h_{s_n}(t_n)) \ .$$

(If you compare this definition with Definition 5.4, you'll notice that this is an inductive definition of the functions h_s. Note also that, since there is one family of functions for each Σ-model M, we should probably write something like 'h^M' instead of 'h'; but '$h^M_{s_1}(t_1)$' looks much uglier than '$h_{s_1}(t_1)$', so we'll sacrifice precision for aesthetics — except in the following exercise.)

Exercise 5.11 *Following Definition 5.2, we defined models R, Q and S for the signature of* BINARY-NUMERALS*; Exercise 5.7 asked you to define a model T of the same signature. In each of those models, interpret the following terms (i.e., for each of the terms t, evaluate* $h^R_{\mathsf{BinaryNumeral}}(t)$*,* $h^Q_{\mathsf{BinaryNumeral}}(t)$*,* $h^S_{\mathsf{BinaryNumeral}}(t)$*, and* $h^T_{\mathsf{BinaryNumeral}}(t)$*):*

1. *1 0*
2. *1 1*
3. *1 1 0 1*

Exercise 5.12 *For any signature* Σ*, what will be the interpretation of any* Σ*-term in the Lazy-progammer model* L_Σ*?*

Exercise 5.13 *Let* Σ *be the signature of* BOOLEANS*, and let* N *be the* Σ*-model defined by*

$$\begin{aligned} N_{\mathsf{Bool}} &= \{0, 1, 2, \ldots\} \\ N_{\mathsf{true}} &= 1 \\ N_{\mathsf{false}} &= 0 \\ N_{\mathsf{not}}(x) &= x + 1 \quad \textit{for all } x \in N_{\mathsf{Bool}} \\ N_{\mathsf{or}}(x, y) &= x + y \quad \textit{for all } x, y \in N_{\mathsf{Bool}} \\ N_{\mathsf{and}}(x, y) &= x \times y \quad \textit{for all } x, y \in N_{\mathsf{Bool}} \end{aligned}$$

For each of the terms in Example 4.4.1, say what values they are given by h^N*.*

Interlude. homomorphisms and initial algebras: the inductive definition of h^M is the proof of initiality

---

Let's summarise what we've covered so far, and then make one point about the semantics of Maude. A Maude specification declares sorts and operations (we're ignoring the equations that might be included in a specification). These declarations can be viewed in a rather abstract way as a signature (S, O): a set of sort-names, S, and a collection of operation-names, O (but remember that each operation has an arity, that's why O is a *family* of sets $O_{w,\,s}$, where w is the list of argument sorts, and s is the result sort). Schematically, a Maude specification (with no equations) looks like

fmod SOME-NAME **is**

$\Sigma = (S, O)$

endfm

Models of such a specification are the Σ-models: these provide a concrete representation for the abstract data values, by giving implementations of the constants and operations that are declared in the specification. Alternatively, we could say that models interpret sort-names as sets (the 'carrier sets'), and interpret constants as elements and operations as functions. This interpretation extends to terms: each model says how any term is to be evaluated. Every specification has at least one model. we gave two recipes for constructing models for any signature: the Lazy-programmer model and the term algebra. The term algebra seems the most interesting of the two, because it has this rather intriguing property that any term can be interpreted in any model. The examples we looked at suggested that this rather vague notion of interpretation actually had something to do with computation: one model might compute the number represented by a numeral; another model might tell us whether numerals represent odd or even numbers (though weird models, like you were invited to concoct in Exercise 5.7, might not do anything interesting at all).

At this point, we can say why our Maude specifications of the syntax of SIMPLE are exactly what we want. A Maude functional module specifies an 'initial' algebra. We'll say exactly what that means in Section 5.4, but when a module contains no equations, the term algebra is initial. The modules EXPRESSION, PROGRAM, etc., therefore specify terms: i.e., the expressions, boolean expressions, and programs that we can write down in the programming language.

We've been ignoring equations so far; we'll look at those in Section 5.4. We've also ignored subsort declarations; we'll look at those now.

5.3 Order-sorted Specifications

When you answered Exercise 5.6, you may — and should — have wondered what happened to the declaration

subsort BinaryDigit < BinaryNumeral .

Well, we just ignored it for the sake of simplicity. But now we've got to grips with the notions of signature and model, let's get complicated. The truth is that a Maude specification doesn't just give us a set of sorts; those sorts are *ordered*: some sorts are subsorts of other sorts. For example, in the BINARY-NUMERAL-EXPRESSIONS specification, BinaryDigit is a subsort of BinaryNumeral, and BinaryNumeral is a subsort of Expression. Generally, a Maude specification gives us a set of sorts, *and a subsort relation on those sorts*, and a collection of operations. So a signature isn't just a pair (S, O), but looks like $((S, \leq), O)$, where $\leq$ is the subsort relation on the sorts S. In the module BINARY-NUMERAL-EXPRESSIONS, we have $S = \{\mathsf{BinaryDigit}, \mathsf{BinaryNumeral}, \mathsf{Expression}\}$ and the subsort declarations tell us that

$$\mathsf{BinaryDigit} \leq \mathsf{BinaryNumeral} \tag{5.10}$$

and

$$\mathsf{BinaryNumeral} \leq \mathsf{Expression} \tag{5.11}$$

The guiding principle for subsorts is this: everything in the subsort is also in the supersort. So, for example, every BinaryDigit is also a BinaryNumeral, and every BinaryNumeral is also an Expression. It follows that every BinaryDigit is also an Expression. We know that 1 is a BinaryDigit, so it is also a BinaryNumeral. It is also, therefore, an Expression. We say that the subsort relation is *transitive*: this means that if a sort s_1 is a subsort of a sort s_2 ($s_1 \leq s_2$), and if s_2 is a subsort of a sort s_3 ($s_2 \leq s_3$), then s_1 is a subsort of s_3 ($s_1 \leq s_3$).

To capture the notion of a Maude specification that declares sorts, operations, *and* subsorts, we need

Definition 5.7 *An **order-sorted signature** is a pair $((S, \leq), O)$, where (S, O) is a signature (as in Definition 5.1), and $\leq$ is a transitive relation on S.*

We want signatures to be an abstract way of capturing all the information given in a Maude specification. If we have a Maude specification, we extract the signature from it by taking S to consist of all the sorts declared in the specification, and $O_{w,s}$ to consist of all the operations of arity (w, s). The subsort relation should be determined by all the subsort declarations in the specification, but there is nothing transitive about those declarations in themselves. From

subsort BinaryDigit < BinaryNumeral .
subsort BinaryNumeral < Expression .

we still need to make some sort of jump to get

subsort BinaryDigit < Expression .

Now it might be clear to you how to make that jump: just take all the *consequences* of the subsort declarations, and that's exactly right. If you're happy with that, then fine; however, the notion of consequence is interesting so you might want to go ahead and read the following

---

Interlude. relations as sets of pairs; functions as relations; partial orders; least po extending a relation — and inductive definitions

---

We've changed our definition of signature so as to take subsort declarations into account; now we need to update our definition of model so that subsorts are treated in an appropriate way.

Definition 5.8 *Let $\Sigma = ((S, \leq), O)$ be an order-sorted signature. A Σ-model is an (S, O)-model M such that conditions (1)–(3) of Definition 5.2 hold, and*

4. if $s_1 \leq s_2$, then $M_{s_1} \subseteq M_{s_2}$

This definition imposes an extra constraint on what it means to be a model of an order-sorted signature (this is why we've numbered it (4), as an extra requirement on the three requirements in Definition 5.2). Remember the Σ^{B}-model R introduced after Definition 5.2? That was an example of an order-sorted model, because we have

$$\mathsf{BinaryDigit} \leq \mathsf{BinaryNumeral}$$

and Equations (5.1) and (5.2) tell us that $R_{\mathsf{BinaryDigit}} \subseteq R_{\mathsf{BinaryNumeral}}$, just as Definition 5.8(4) requires.

Exercise 5.14 *Check that the model S of Exercise 5.7 is an order-sorted Σ^{B}-model. Check also that the Lazy-programmer model is an order-sorted model of any order-sorted signature.*

This exercise tells us that any order-sorted signature has at least one model: the lazy-programmer model. But the term algebra is *not* an order-sorted model. According to Definition 5.4, $\mathsf{0}$ is a term of sort BinaryDigit, but it's not a term of sort BinaryNumeral, as required by Definition 5.8(4). Why not? Well, BINARY-NUMERALS doesn't declare any constants of sort BinaryNumeral (apart from implicitly, via the subsort declaration — check your answer to Exercise 5.6), so Definition 5.4(1) doesn't apply; and since $\mathsf{0}$ is a constant, Definition 5.4(2) doesn't apply either, so there just isn't anything that tells us $\mathsf{0}$ is a term of sort BinaryNumeral. So $\mathsf{0} \in (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryDigit}}$, but $\mathsf{0} \notin (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryNumeral}}$, so we don't have $(T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryDigit}} \subseteq (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryNumeral}}$, as we would need for the term algebra to be an order-sorted model.

How to fix this? Simple: just say that all terms of a subsort are also terms of the supersort.

Definition 5.9 *For an order-sorted signature $\Sigma = ((S, \leq), O)$, the term algebra is defined as in Definitions 5.4 and 5.5, but we add the following to the definition of terms:*

3. for all sorts $s_1, s_2 \in S$, if $s_1 \leq s_2$ then $(T_\Sigma)_{s_1} \subseteq (T_\Sigma)_{s_2}$.

This extra clause adds to the definition of terms of sort s_2: they include all the terms of sort s_1. For example, since

$$\mathsf{BinaryDigit} \leq \mathsf{BinaryNumeral} \ ,$$

Definition 5.9(3) tells us that

$$(T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryDigit}} \subseteq (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryNumeral}}$$

i.e., all terms of sort BinaryDigit are also terms of sort BinaryNumeral. Thus, since $\mathsf{0} \in (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryDigit}}$, it follows (from this definition) that $\mathsf{0} \in (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryNumeral}}$, and we get $\mathsf{1} \in (T_{\Sigma^{\mathsf{B}}})_{\mathsf{BinaryNumeral}}$ in a similar way. And that's exactly what we want

---

Interlude. why this definition works; terminology: many-sorted and order-sorted; other approaches (membership logic, action semantics); also: kinds

---

5.4 Equations: How Things Behave

So far, we've looked at signatures of specifications. Basically, these are good for writing terms, and terms represent elemenents in models. Models are the semantics of specifications — the possible implementations — but the relationship between specifications and models seems rather static. Earlier, we said that terms could be thought of as problems: for example, a term `102 * 98` is a problem whose solution is the term `9,996`. Algorithms, such as the 'long multiplication' algorithm you learnt at school, are ways of going from problems to their solutions. Equations are another way of relating problems to solutions, and that's what we'll look at in this section. As far as functional programming is concerned, equations can be used to *define* functions, and we'll return to this computational aspect of equations in Section 5.4.1.

Here's an example of an equation, one we saw in the previous chapter:

```
eq  true and Y = Y .
```

Maude syntax requires that equations be written in a particular form, with particular keywords ('`eq`', '`=`', spaces, and of course a full-stop), but the important components of an equation are two terms: the left-hand side (in this example, `true and Y`) and the right-hand side (`Y`). Both left- and right-hand sides may contain variables (in this example, `Y`), so when we refer to an equation, it makes sense to include the variable declarations and that's what we'll do from now on. So here's the official version of the equation from module BOOLEANS:

```
var  Y : Bool .
eq   true and Y = Y .
```

The important parts, the left- and right-hand sides, are still there, but now we've also got the variable declaration in there, so we know how to make sense of the term `true and Y`. Actually, if we want to make sense of the left- and right-hand sides, we should also include the declarations of the constant `true` and the operation `and`, so from now on, we'll talk about Σ-equations, where Σ is a signature (e.g., the signature of BOOLEANS). So, for example, let Σ be the signature of BINARY-NUMERAL-SEMANTICS; here's a Σ-equation:

```
var  N : BinaryNumeral .
var  D : BinaryDigit .

eq   [[ N D ]] = 2 * [[ N ]] + [[ D ]] .
```

In this equation, the problem — i.e., the left-hand side — is to work out the value of a numeral of the form `N D`. The right-hand side says how to solve the problem: find the value of `N`, multiply it by `2`, and add the value of `D` to the result.

That's the interesting thing about equations: they relate solutions to problems. But before we can give that interesting stuff the full attention it deserves, we have to be a bit more precise about what equations are. First up, what exactly are variables? (Sneak preview: they're just constants.)

Variables are just constants. In the equations we looked at above, `Y`, `N`, and `D` are just symbols that are used to make up the terms that are the left- and right-hand sides of the equations. According to our definition of terms, terms are built up from constants and operations (with brackets and commas around subterms), so we can't have terms with constants that haven't been declared:

```
true and someNameThatHasntBeenDeclared
```

just isn't a term. However, if we say that `Y` is a constant, then both `true and Y` and `Y` are terms. These are the left- and right-hand sides of the first equation above.

So that's why variables are declared in Maude: a variable declaration introduces a new constant that can be used in writing the terms in the left- and right-hand sides of equations. Let's say all of this a bit more formally. First, constants:

Definition 5.10 *A **ground signature** is a signature (S, X) that contains only constants; that is, $X_{w,\,s} = \emptyset$ whenever $w \neq [\,]$.*

Note that the set $X_{[\,],\,s}$ is the set of all the variables of sort s; for the BINARY-NUMERALS-SEMANTICS example, where our only variables were

```
var  D : BinaryDigit .
var  N : BinaryNumeral .
```

we have

$$\begin{aligned} X_{[\,],\,\mathsf{BinaryDigit}} &= \{\mathsf{D}\} \\ X_{[\,],\,\mathsf{BinaryNumeral}} &= \{\mathsf{N}\} \end{aligned}$$

and for our specification of unary numerals with addition (see page 46) where our only variables were

```
vars M N : Nat .
```

we have

$$X_{[\,],\,\mathsf{Nat}} = \{\mathsf{M}, \mathsf{N}\}$$

Now, variables as constants:

Definition 5.11 *Let $\Sigma = (S, O)$ be a signature. A **signature of Σ-variables** is a ground signature (S, X) that is disjoint from Σ.*

In a bit more detail: we're assuming we have a Maude specification, and that specification will have a signature — let's call it $\Sigma = (S, O)$. That Maude specification may declare some variables that will be used in its equations. All of those variables are constants, and that's why we say it's a ground signature. All of those variables must have a sort, and that sort must have been declared in the spec — that's why we say the ground signature must use S: we don't want variable declarations like

```
var  V : SomeSortThatHasn'tBeenDeclared .
```

Definition 5.12 *Let $\Sigma = (S, O)$ be a signature, and let (S, X) be a signature of Σ-variables. We write $T_\Sigma(X)$ for the algebra of terms with variables built from the operations in Σ and the constants in X.*

To be precise, $T_\Sigma(X) = T_{\Sigma \cup X}$, where $\Sigma \cup X$ is the signature defined by $(\Sigma \cup X)_{w,\,s} = O_{w,\,s} \cup X_{w,\,s}$.

Definition 5.13 *Let $\Sigma = (S, O)$ be a signature. A **Σ-equation** is a triple (X, l, r), where*

- *X is a ground signature disjoint from O, and*
- *l and r are $\Sigma(X)$-terms of the same sort (i.e., $l, r \in T_\Sigma(X)_s$ for some $s \in S$).*

We usually write $(\forall X)\ l = r$ instead of (X, l, r).

Although we said that variables are constants, we don't declare them in Maude with the keyword op; we use the keyword var to declare variables because they're used in a different way: they represent arbitrary values. For example, in the equation

```
eq   true and Y = Y .
```

from the module BOOLEANS, the variable Y stands for an arbitrary Bool value — so it's standing for either true or false, right? Well, sort of.

We can take a very syntactic view of equations, in which case, Y does stand for either true or false; this syntactic view lets us do term-rewriting, and gives us an operational semantics for Maude, as we see in the next subsection. Later on, in Section 5.4.2, we'll take a more semantical view of equations, and look at the denotational semantics of Maude.

---

move defn of eqnl th here

5.4.1 Term-rewriting: Maude's Operational Semantics

In Chapter 4 we saw that Maude uses equations to simplify terms when we use the reduce command; in this section we'll see exactly what happens when Maude simplifies terms.

Let's revisit the caveman notation of Section 4.5.2, which had this signature:

```
sort  Nat .
op    0 :  → Nat .
op    s : Nat → Nat .
op    _ + _ : Nat Nat → Nat .
```

and these equations:

```
vars  M N : Nat .
eq    0 + N = N .
eq    s(M) + N = s(M + N) .
```

How does this apply to a term such as s(s(0)) + s(s(0))? The first thing to note is that the first equation doesn't apply to this term, because the term to the left of the + isn't 0. The second equation *does* apply, because the term to the left of the + has the form s(M) — in this case, M is s(0). Note how M is simply functioning as a name for a term: whatever term is the argument to s on the left of the + symbol. In the same way, N is simply functioning as the name for whatever term is on the right of the +. In the case of our term s(s(0)) + s(s(0)), N is naming the term s(s(0)). The equation says that this term is equal to s(M + N), and since M is naming s(0) and N is naming s(s(0)), this means

$$\mathsf{s(s(0))} + \mathsf{s(s(0))} \quad = \quad \mathsf{s(\ s(0) + s(s(0))\ )}$$

In the process of matching the left-hand side of the equation to the term we want to rewrite, we associate the variables in the left-hand side with terms (e.g., M with s(0); N with s(s(0))). You can think of this association as a table:

M	N
s(0)	s(s(0))

or you can think of it as a function, f, mapping the variables to terms:

$$f(\mathsf{M}) \quad = \quad \mathsf{s(0)} \tag{5.12}$$

$$f(\mathsf{N}) \quad = \quad \mathsf{s(s(0))}\ . \tag{5.13}$$

In general, we may have variables of different sorts, and each variable should be matched with a term of the same sort, so we need a family of functions, f_s, for each sort s, mapping variables of sort s to terms of sort s. Such a family of functions is called an *assignment to the variables*[3] For some extra generality, we can even allow these to assign terms with variables:

Definition 5.14 *Let $\Sigma = (S, O)$ be a signature, and let X and Y be signatures of Σ-variables. An **assignment to the variables in** X is a family of functions $f_s : X_{[\,],\,s} \to T_\Sigma(Y)_s$ for $s \in S$.*

For the sake of brevity, we usually just write $f : X \to T_\Sigma(Y)_s$ to indicate that f is an assignment to the variables in X.

Usually, we want to rewrite terms that *don't* contain variables, so most assignments to variables will have $Y = \emptyset$. For example, recall the equation from BINARY-NUMERALS-SEMANTICS in Section 4.5.4:

```
var  N : BinaryNumeral .
var  D : BinaryDigit .

eq   [[ N D ]] = 2 * [[ N ]] + [[ D ]] .
```

And suppose we want to use this equation to simplify [[1 0 1]]. The signature of variables in this example — let's call it X — has

$$\begin{aligned} X_{[\,],\,\mathsf{BinaryNumeral}} &= \{\mathsf{N}\} \\ X_{[\,],\,\mathsf{BinaryDigit}} &= \{\mathsf{D}\} \end{aligned}$$

and the assignment to these variables that we're interested in for this example is $f : X \to T_\Sigma(\emptyset)$ with

$$f_{\mathsf{BinaryNumeral}}(\mathsf{N}) = \mathsf{1\ 0} \tag{5.14}$$

$$f_{\mathsf{BinaryDigit}}(\mathsf{D}) = \mathsf{1} \tag{5.15}$$

When we say that the term [[1 0 1]] *matches* the equation, we mean that when we replace the variables in the left-hand side with the terms that f assigns to them, then we get the term [[1 0 1]] itself. The following gives a formal definition of how we substitute terms for variables (again, for extra generality, we allow the terms we substitute to themselves contain variables).

Definition 5.15 *Let $\Sigma = (S, O)$, be a signature and let f be an assignment to the variables in X; i.e., f is a family of functions $f_s : X_{[],\,s} \to T_\Sigma(Y)_s$ for $s \in S$. Define the functions $h^f_s : T_\Sigma(X)_s \to T_\Sigma(Y)_s$ by*

1. *for variable $v \in X_{[],\,s}$, $h^f_s(v) = f_s(v)$,*
2. *for constant $c \in \Sigma_{[],\,s}$, $h^f_s(c) = c$, and*
3. *for operation $o \in \Sigma_{w,\,s}$, with $w = s_1 s_2 \cdots s_n$ and $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1..n)$, $h^f_s(t_1, \ldots, t_n) = o(h^f_{s_1}(t_1), \ldots, h^f_{s_n}(t_n))$*

Maude allows us to specify the syntax we want using underscores; Definition 5.15 assumes we're using default prefix-and-brackets syntax, while the left-hand side [[N D]] makes heavy use of underscores. If we put this left-hand side into standard prefix-and-brackets syntax, it looks like [[_]](_ _ (N,D)). Let's use this (rather ugly) form of the left-hand side, and the assignment f defined above (equations (5.14) and (5.15)), to illustrate this definition. Starting with

$$h^f_{\mathsf{Int}}([[\_\,]](\_\ \_\ (\mathsf{N},\mathsf{D})))\ ,$$

[3]This has nothing at all to do with assignments like `'x := 0`. Variables in imperative programming languages are names, but they're not *just* names; they're names *for* storage addresses — typically, addresses in working memory where the *values* assigned to the variables are stored. In contrast to this, variables in Maude, and in logical languages such as Predicate Logic, really are just names, as should be obvious from Definitions 5.10 and 5.11. Variables in imperative programming languages store particular values, whereas variables in Maude stand for arbitrary values.

Definition 5.15(3) tells us this is equal to

$$[[\_\,]](h^f_{\mathsf{BinaryNumeral}}(\_\ \_\ (\mathsf{N},\ \mathsf{D})))\ .$$

Once again, Definition 5.15(3) tells us this is in turn equal to

$$[[\_\,]](\_\ \_\ (h^f_{\mathsf{BinaryNumeral}}(\mathsf{N}),\ h^f_{\mathsf{BinaryDigit}}(\mathsf{D})))\ .$$

Now we can use Definition 5.15(1) twice to get

$$[[\_\,]](\_\ \_\ (f_{\mathsf{BinaryNumeral}}(\mathsf{N}),\ f_{\mathsf{BinaryDigit}}(\mathsf{D})))\ .$$

Finally, equations (5.14) and (5.15) tell us that this is equal to

$$[[\_\,]](\_\ \_\ (1\ 0,\ 0))\ ,$$

i.e., [[1 0 1]]. Putting all this into Maude's mixfix notation, and omitting all those distracting subscripts, what we've done is:

$$h^f([[\ \mathsf{N}\ \mathsf{D}\ ]]) = [[\ h^f(\mathsf{N})\ h^f(\mathsf{D})\ ]] = [[\ f(\mathsf{N})\ f(\mathsf{D})\ ]] = [[\ 1\ 0\ 1\ ]]\ .$$

Exercise 5.15 *Now do the same thing with the right-hand side of the equation: for the same f, simplify h^f_{Int}(2 * [[N]] + [[D]]).*

All of this shows us exactly what is going on when Maude rewrites the term [[1 0 1]] to the term 2 * [[N]] + [[D]]: first, Maude finds an assignment to the variables, f, such that applying h^f to the left-hand side of the equation gives the term that is to be rewritten, and does the rewriting by replacing that term with the result of applying h^f to the right-hand side of the equation.

In general, a term t rewrites to a term t' using an equation $(\forall X)\ l = r$ if there is an assignment, f, to the variables in X such that $t = h^f(l)$ and $h^f(r) = t'$.

Exercise 5.16 *Using the f defined in (5.12) and (5.13) on p. 65, show how Maude rewrites s(s(0)) + s(s(0)) to s(s(0) + s(s(0))). Do this twice, using prefix-with-parentheses notation, and with Maude's mixfix notation (for _ +_).*

So what does Maude do once she's rewritten s(s(0)) + s(s(0)) to s(s(0) + s(s(0)))? She should be able to go on to rewrite s(0) + s(s(0)) to s(0 + s(s(0))) ...

Exercise 5.17 *Check this. While you're at it, check that 0 + s(s(0)) rewrites to s(s(0)).*

...and that should mean that s(s(0) + s(s(0))) rewrites to s(s(0 + s(s(0)))). In this case, the equation is being applied to a *subterm* of the term. The subterm being rewritten is s(0) + s(s(0)), and this takes place in the *context* s(_), by which we mean that the underscore marks the location of the subterm that is being rewritten. A context like this is just a term that contains the symbol '_'.

Definition 5.16 *A **context** is a term that contains the symbol '_'. We usually write $c[\_\,]$ for contexts, and for a term $t \in (T_\Sigma)_s$, we write $c[t]$ for the result of replacing '_' with t.*

For example, if $c[\_\,]$ is s(_), then $c[0]$ is s(0), $c[s(0)]$ is s(s(0)), and $c[s(0) + s(s(0))]$ is s(s(0) + s(s(0))).

Incidentally, this gives us a way of defining the notion of *subterm*. a term u is a subterm of a term t iff there is a context c such that $t = c[u]$.

Example 5.2 *s(s(0)) is a subterm of s(0 + s(s(0))) because of the context s(0 + _).*

Exercise 5.18 *Why is 1 0 a subterm of 1 0 1, and why is 0 1* not *a subterm of 1 0 1? Find all the contexts that show that s(0) is a subterm of s(s(0)) + s(s(s(0)) + s(0 + s(0))).*

Exercise 5.19 *The 'is a subterm of' relation is reflexive: every term t is a subterm of itself. What context shows this?*

Interlude. A context is just a term that contains the underscore symbol, '_ ', and since variables are just new symbols that can be used in terms, it seems a shame not to re-use some of that technical machinery. If we were to be more precise about contexts, we should say:

- firstly, it only makes sense to talk about terms if we have some signature that we can use to build terms with, so let's say we're talking about Σ-contexts, where $\Sigma = (S, O)$ is some signature;
- secondly, terms have to be well-formed — if the underscore symbol occurs in a term, it has to be at some point where a term of a certain sort is expected (and this sort should be the same as the sort of the left- and right-hand sides of the equation that we're looking to apply), so the underscore symbol could be thought of as a constant of that sort.

So if $s \in S$ is a sort, let's write $\underline{s}$ for the ground signature that contains only the underscore symbol, which is of sort s (i.e., $\underline{s}_s = \{\_\}$ and $\underline{s}_x = \emptyset$ if $x \neq s$.). Then a context for sort s is a term in $T_\Sigma(\underline{s})$.

Exercise 5.20 *Now use Definition 5.15 to explain what precisely is meant by the notation $c[t]$ of Definition 5.16.*

Maude rewrites the term s(s(0) + s(s(0))) by rewriting s(0) + s(s(0)) to s(0 + s(s(0))) as above, *within the context* s(_). The result is that s(s(0) + s(s(0))) is rewritten to s(s(0 + s(s(0)))). Maude will then rewrite this term to s(s(s(s(0)))) by rewriting 0 + s(s(0)) to s(s(0)) within the context s(s(_)).

To summarise, the sequence of rewrites is given below; at each step, we underline the subterm that is being rewritten.

$$\begin{aligned} \underline{\mathsf{s(s(0))} + \mathsf{s(s(0))}} &= \mathsf{s}(\underline{\mathsf{s(0)} + \mathsf{s(s(0))}}) \\ &= \mathsf{s(s}(\underline{\mathsf{0} + \mathsf{s(s(0))}})) \\ &= \mathsf{s(s(s(s(0))))} \end{aligned}$$

Exercise 5.21 *For each step above, say:*

1. *which equation is being used;*
2. *what assignment to the variables is being used; and*
3. *the context that identifies the subterm being rewritten.*

You can check your answers to parts (1) and (2) of this exercise using Maude and the command set trace on . If you type this command before a reduction, Maude will show the equation and assignment to the variables that are used at each step. You can toggle this feature off with the command set trace off . (You need the dot at the end of these commands.)

Exercise 5.22 *Experiment with the command* set trace whole on .

In general, what Maude does when she rewrites a term is this:

1. find a subterm of the term that matches the left-hand side of an equation: this involves finding a context that identifies the subterm, and finding an assignment, f, to the variables in the equation, such that the subterm is equal to the result of applying h^f to the left-hand side of the equation; then
2. replace that subterm with the result of applying h^f to the right-hand side of the equation.

Let's suppose we have an equation $(\forall X)\ l = r$, and let's suppose we have a term t. The term t matches the equation if we can find an assignment to the variables in X such that $t = h^f(l)$. But more generally, maybe a *subterm* of t matches the equation: maybe $t = c[u]$ (i.e., u is a subterm of t) and u matches the equation; i.e., $u = h^f(l)$ for some assignment f, which means that $t = c[h^f(l)]$. When Maude applies the

equation, she replaces the subterm u with the right-hand side of the equation, with all variables replaced by the terms they match in the left-hand side. In other words, u is replaced with $h^f(r)$, and instead of the term $c[h^f(l)]$, we have the term $c[h^f(r)]$. This is what we call *term rewriting*, and we say the term $c[h^f(l)]$ rewrites to $c[h^f(r)]$. We write $t \to_E t'$ to indicate that the term t rewrites to the term t', using some equation in the set E. Of course, we're usually interested in sequences of rewrites, such as the sequence that rewrites $\mathsf{s(s(0))} + \mathsf{s(s(0))}$ to $\mathsf{s(s(s(s(0))))}$: we write $t \to^*_E t'$ to indicate that the term t rewrites in zero or more steps to t', using equations from the set E for each step. This means there are some (zero or more) intermediate terms:

$$t \to_E t_1 \to_E t_2 \to_E \cdots \to_E t_n \to_E t' \ .$$

Allowing zero rewriting steps means that the relation $\leftarrow_E$ is a little bit like equality: every term is related to itself. This makes sense because rewriting uses the *equations* in a specification. Maude, however, only applies equations left-to-right. We can get a symmetric relation on terms by allowing equations to be applied right-to-left. This is useful, for example, in showing that $\mathsf{s(0)} + \mathsf{s(s(0))} = \mathsf{s(s(0))} + \mathsf{s(0)}$ is a consequence of the equations for unary arithmetic. Define $t \leftrightarrow_E t'$ to mean *either* $t \to_E t'$ *or* $t' \to_E t$, and define $t \overset{*}{\leftrightarrow}_E t'$ to mean that t is related to t' by some number of steps using $\leftrightarrow_E$. This means there are zero or more intermediate terms

$$t \leftrightarrow_E t_1 \leftrightarrow_E t_2 \leftrightarrow_E \cdots \leftrightarrow_E t_n \leftrightarrow_E t' \ .$$

For example, we might have

$$t \ {}_E\!\leftarrow \ t_1 \ \to_E \ t_2 \ \to_E \ t_3 \ {}_E\!\leftarrow \ t_4 \ \to_E \ t' \ .$$

Exercise 5.23 *Show that* $\mathsf{s(0)} + \mathsf{s(s(0))} \overset{*}{\leftrightarrow}_E \mathsf{s(s(0))} + \mathsf{s(0)}$*, where E is the set of equations defining addition on unary numerals.*

Let's try and summarise all this. Term-rewriting is the process of 'simplifying' terms using equations Term-rewriting is how Maude uses equations to do computations

5.4.2 Satisfaction: Maude's Denotational Semantics

Remember our discussion of the specification BOOLEANS and especially the equation

```
eq  true and Y = Y .
```

If we think of this equation as a rewrite rule, then the variable Y 'stands for' any term of sort Bool, in the sense that it can match with any such term, as explained in the previous subsection.

On page 55 we described the 'reasonable' model R with $R_{\mathsf{Bool}} = \{true, false\}$. In this model, Y stands for either true or false, but we've seen that Maude specifications can have all sorts of weird and wonderful models. Remember that a model is a possible implementation of a specification. While we were ignoring equations and concentrating on signatures, models simply consisted of carrier sets and implementations of the operations as functions that worked with those carrier sets. Any function would do, so long as it had the right type. The purpose of equations is to be a bit more discriminating about what is an acceptable implementation: equations tell us (or the programmers who have to implement a spec) what the operations *do*; they give the programmer who implements the operations some rules about what his implementation must do. If I implement BOOLEANS using keywords `true` and `false`, then the equation above tells me that I have to make sure I implement and in such a way that if the arguments are `true` and `true` then the result must be `true`, and if the arguments are `true` and `false` then the result must be `false`.

But suppose I prefer C's approach to Booleans, and I want to implement Bool using the integers, with false implemented as 0, and true implemented as 1 (or any other non-zero integer). The equation still imposes a constraint on how I implement and: if the first argument is 1 (or non-zero), then the result must be the same as the second argument. According to the equation,

```
int and(int x, int y) {
    return x * y;
}
```

would be an acceptable implementation (because `1 * y` = `y`), but

```
int and(int x, int y) {
    return x + y;
}
```

would not be an acceptable implementation (because `1 + y` $\neq$ `y`).

What Y stands for depends upon how the programmer chooses to represent Bool. Let's throw you in at the deep end. Here's a Maude spec that defines unary numerals with addition and multiplication:

```
fmod ARITHMETIC is
  sort Nat .
  op 0 : -> Nat [ ctor ] .
  op s : Nat -> Nat [ ctor ] .
  ops (_ + _) (_ * _) : Nat Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
endfm
```

and here are four Σ-models (i.e., four possible implementations of the signature of this spec):

$A_{\mathsf{Nat}} = \{0, 1, 2, \ldots\}$
$A_{\mathsf{0}} = 0$
$A_{\mathsf{s}}(m) = m + 1$
$A_{+}(m, n) = m + n$
$A_{*}(m, n) = m \times n$

$B_{\mathsf{Nat}} = \{0, 1\}$
$B_{\mathsf{0}} = 0$
$B_{\mathsf{s}}(m) = 1 - m$
$B_{+}(m, n) = max(m, n)$
$B_{*}(m, n) = min(m, n)$

$C_{\mathsf{Nat}} = \{true, false\}$
$C_{\mathsf{0}} = false$
$C_{\mathsf{s}}(m) = \neg m$
$C_{+}(m, n) = m \vee n$
$C_{*}(m, n) = m \wedge n$

$D_{\mathsf{Nat}} = \{true, false\}$
$D_{\mathsf{0}} = true$
$D_{\mathsf{s}}(m) = m$
$D_{+}(m, n) = m \wedge n$
$D_{*}(m, n) = \neg m \vee n$

Exercise 5.24 (Spot the model) *Which models satisfy which equations?*

Exercise 5.25 *Give a model E with $E_{\mathsf{Nat}} = \{true, false\}$ that satisfies all of the equations in ARITHMETIC.*

You probably found that the model A satisfied all the equations, and that the model C satisfied the equation

```
eq 0 * N = 0 .
```

because 0 is implemented as *false* and * is implemented as conjunction, and $false \wedge \mathsf{N} = false$, whether N is *true* or *false*. Maybe you also spotted that model D doesn't satisfy that equation: D implements 0 as *true* and implements * as implication, so the left-hand side becomes $\neg true \wedge \mathsf{N}$, which is equal to *false* if N is *false*, but it's *not* equal to *false* when N is *true*. This gives us a *counter-example* that shows the equation isn't satisfied: when we put the value *true* in for N and evaluate the left- and right-hand sides of the equation, we get different results.

Note, though, that we evaluate the two sides of the equation according to the particular implementation of the model, so we're using something like the functions of Definition 5.6 that told us how to evaluate terms in models, and replacing the variables in the equation with all the possible values from the model's carrier sets. For example, we might check whether model D satisfies the equation

eq s(M) + N = s(M + N) .

by going through all the possible assignments to M and N, and evaluating both sides of the equation (recall that D implements s as the identity function, and + as conjunction):

M	N	s(M) + N	s(M + N)
true	*true*	*true*	*true*
true	*false*	*false*	*false*
false	*true*	*false*	*false*
false	*false*	*false*	*false*

We can see that both sides are the same whatever values (*true*/*false*) we choose for M and N. So D does indeed satisfy the equation.

When we talk about choosing values in for the variables, we just mean a function from variables to values. Each row of the table above is just one of all four possible functions from $\{\mathsf{M}, \mathsf{N}\}$ to $\{true, false\}$. Row 1 is the function f_1, with

$$f_1(\mathsf{M}) = true = f_1(\mathsf{N}) \ ,$$

Row 2 is the function f_2, with

$$f_2(\mathsf{M}) = true \quad \text{and} \quad f_2(\mathsf{N}) = false \ ,$$

and so on. In general, when we have variables of different sorts, we want a family of functions $f_s : X_{[],s} \to M_s$ for each $s \in S$. That is, for each sort s, f_s maps the variables of sort s (in $X_{[],s}$) to values of the same sort (in the carrier set M_s). If we have such a family of functions, then we can interpret not just any term, as in Definition 5.6, but we can interpret any term that contains variables:

Definition 5.17 *Let $\Sigma = (S, O)$, be a signature, let M be a Σ-model, and let f be a family of functions $f_s : X_{[],s} \to M_s$. Define the functions $h^f_s : T_\Sigma(X)_s \to M_s$ by*

- *for variable $v \in X_{[],s}$, $h^f_s(v) = f_s(v)$,*
- *for constant $c \in \Sigma_{[],s}$, $h^f_s(c) = M_c$, and*
- *for operation $o \in \Sigma_{w,s}$, with $w = s_1 s_2 \cdots s_n$ and $t_i \in T_\Sigma(X)_{s_i}$ $(i = 1..n)$,*

$$h^f_s(t_1, \ldots, t_n) = M_o(h^f_{s_1}(t_1), \ldots, h^f_{s_n}(t_n)) \ .$$

Exercise 5.26 *For each row in the table above, say what the function $f_{\mathsf{Nat}} : X_{[],\mathsf{Nat}} \to$ is, then apply h^f_{Nat} to the left- and right-hand sides of the equation to check the entries in Columns 2 and 3 of the table.*

Now we can be precise about what it means for a model to satisfy an equation:

Definition 5.18 *Let $\Sigma = (S, O)$ be a signature, let $e = (\forall X) l = r$ be a Σ-equation, and let M be a Σ-model. M **satisfies** e (written $M \models e$) iff $h^f_s(l) = h^f_s(r)$ for all families of functions $f_s : X_{[],s} \to M_s$ for $s \in S$.*

We introduced signatures as an abstract way of bundling up all the sort an operation declarations in a Maude spec; when we add the equations in the spec, what we get is:

Definition 5.19 *An **equational theory** is a pair (Σ, E), where Σ is a signature and E is a set of Σ-equations.*

*A (Σ, E)-**model** is a Σ-model M such that $M \models e$ for every $e \in E$.*

So an implementation of a Maude spec (a model) is a model of the signature that satisfies all the equations in the spec. For example:

Exercise 5.27 *Check that the Lazy-programmer Model L_Σ is a (Σ, E)-model for any equational theory (Σ, E).*

But hang on — surely we shouldn't encourage lazy programming?! Well, we won't. Although an implementation that satisfies all the equations in a spec is desirable, the Lazy-programmer Model suggests that often it's not enough just to satisfy the equations. What we usually want is an 'initial' model, and we'll say exactly what that is in Section 5.6; before that, we need to explore one more aspect of equations: their consequences.

Imagine you were omniscient. You could look at an equational theory (Σ, E) and see in your mind's eye all of its models. You could also see all the equations that each of those models satisfied. You'd notice some patterns. For example,

No-one is omniscient, but

Definition 5.20 ***Soundness** of equational deduction means that for all signatures Σ, for all sets E of Σ-equations and all Σ-equations e: $\models Ee$ if $E \vdash e$.*

Definition 5.21 ***Completeness** of equational deduction means that for all signatures Σ, for all sets E of Σ-equations and all Σ-equations e: $\models Ee$ implies $E \vdash e$.*

5.5 Rewrite Logic and Equational Logic

What is a logic?

Even as we begin, there's something odd about that question: why not ask 'what is logic' — why say '*a* logic' rather than just 'logic'? Surely logic is just logic?

For a long time, logic *was* just logic. It all began with Aristotle.

...

Equational logic is the logic of equations. Given a signature, say Σ, the sentences of equational logic are Σ-equations.

- if $e \in E$, then $E \vdash e$
- if $E \vdash (\forall X)\ l = r$, then $E \vdash (\forall Y)\ h^f(l) = h^f(r)$ for any $f\ :\ X \to T_\Sigma(Y)$
- if $E \vdash (\forall X)\ l = r$, then $E \vdash (\forall X)\ r = l$
- if $E \vdash (\forall X)\ t_1 = t_2$ and $E \vdash (\forall X)\ t_2 = t_3$, then $E \vdash (\forall X)\ t_1 = t_3$.
- if $E \vdash (\forall X)\ l = r$, then $E \vdash (\forall X)\ c[l] = c[r]$ for any context c.

Soundness and completeness of equational logic; relate to term-rewriting: equations and rewrite rules, pragmatics

5.6 At last: the meaning of Maude

There are lots of features of Maude that we don't use in this book: have a look at the Maude Manual[4] to see what we've missed out. The Maude that we do use in this book is a very simple language that's geared to specifying abstract data types. You declare sorts and operations, and write equations that say how the operations behave. We've put those three basic elements (sorts, operations and equations) under the microscope in this chapter so let's just rub our eyes, blink a couple of times, and sit back and remind ourselves of the bigger picture. And then we can pull everything together and say what exactly a Maude specification means.

We write a Maude specification in order to specify an abstract data type. An abstract data type consists of a set of data values and some operations that act upon those values. In Maude, we give names to the sets of data values by declaring sorts. Those names

If we have an equational theory (Σ, E), this is meant to specify an abstract data type, and it would be nice if there was a *canonical* model whose carrier sets contained all the abstract data values, and only those values. We don't want these carrier sets to contain random values: they should be the smallest sets containing everything that Σ requires them to contain — and this would have the pleasant side-effect that we could, if necessary, define functions on these carrier sets by induction. In order to be a model of the spec, of course, these carrier sets (and the implementation of the operations) have to satisfy the equations E. On the other hand, we've seen (in Exercise 5.27) that the Lazy-programmer Model does satisfy all equations, but is clearly not what we want as a canonical model of a Maude spec. The Lazy-programmer model has too few elements in the carrier sets: just one, and that one element has to do all the work of implementing the abstract data type, which is clearly pushing things too far.

There are thus two complementary forces driving us towards what an ideal model of a Maude spec should be. On the one hand, the carrier sets shouldn't be too big: they should contain only those elements that Σ says they should. On the other hand, they shouldn't be too small; the Lazy-programmer Model is clearly too small, because everything collapses to one point. Our ideal model should contain everything that Σ requires it to, but should only collapse together the things that E requires to be collapsed together.

When we say 'not too big', we're talking about what the signature Σ requires. Σ contains all the operations of the specification, and these operations let us build terms. Terms can be thought of as programs, or problems that the specification is intended to solve. For example, $\mathsf{s(s(0))} + \mathsf{s(0)}$ represents the problem 'what is the result of adding 2 and 1?' The equations in our specification should tell us that the solution to this problem is 3, or $\mathsf{s(s(s(0)))}$. Or, we can think of $\mathsf{s(s(0))} + \mathsf{s(0)}$ as a program that applies an addition algorithm to the arguments $\mathsf{s(s(0))}$ and $\mathsf{s(0)}$. According to this view, the equations in the specification tell us what the result of running this program should be. Problems, solutions; programs, results: they're all terms, and all built from the operations in the signature. These are the things that the specification describes, and they have to be implemented by the models of the specification. When we say that a model is 'not too big', we mean that its carrier sets contain only things described by the specification. In particular, the carrier sets contain only things that correspond to terms built from the operations in the signature.

Let's give a definition that captures this notion of 'not too big':

Definition 5.22 *Let (Σ, E) be an equational theory. A (Σ, E)-model I* ***has no junk*** *iff: for every sort s and every $a \in I_s$, there is a term $t \in (T_\Sigma)_s$ such that $a = h_s(t)$, where h is the homomorphism $T_\Sigma \to I$ of Definition 5.6.*

This definition says that a model I has no junk if every element a corresponds to some term t. Consider the model A of ARITHMETIC defined by

- $A_{\mathsf{Nat}} = \{0, 1, 2, 3, \ldots\}$
- $A_{\mathsf{0}} = 0$
- $A_{\mathsf{s}}(m) = m + 1$

[4] see URL

- $A_+(m,n) = m + n$
- $A_*(m,n) = m \times n$.

This model has no junk, because 0 corresponds to the term 0, 1 to s(0), 2 to s(s(0)), and so forth.

If we add an element to this model, then we do get junk. For example, let's build a model B that's just like A, but contains some junk. To get junk, we need an element of the carrier set that isn't named by any term, so let's define the carrier set as

$$B_{\mathsf{Nat}} = \{0, 1, 2, 3, \ldots\} \cup \{\texttt{"junk"}\} \ .$$

We've added a string `"junk"` to the carrier set. You probably won't be surprised to learn that this string is going to be the junk element that doesn't correspond to any term. We still need to say how the operations are implemented in this model. The constant 0 is implemented in the same way as in A: $B_0 = 0$. But to define the implementation of s, we need to say what the result is when the argument is a number, and also what the result is when the argument is the junk string.

$$B_{\mathsf{s}}(m) = \begin{cases} m+1 & \text{if } m \text{ is a number} \\ \texttt{"junk"} & \text{if } m = \texttt{"junk"} \end{cases}$$

Similarly, the implementations of + and * add and multiply numbers; if one or both of the arguments are `"junk"`, then the result is `"junk"`:

$$B_+(m,n) = \begin{cases} m+n & \text{if } m,n \in \{0,1,2,\ldots\} \\ \texttt{"junk"} & \text{otherwise} \end{cases}$$

$$B_*(m,n) = \begin{cases} m \times n & \text{if } m,n \in \{0,1,2,\ldots\} \\ \texttt{"junk"} & \text{otherwise} \end{cases}$$

Models with no junk are also called 'reachable'; the idea is that each element a is 'reached' via the term t such that $h(t) = a$. The module NATURALS of Section 4.5.2 specified the natural numbers using only the constant 0 and the successor operation s. Starting at 0, we can 'reach' all the numbers in turn by repeatedly applying the successor operation: applying it once, we reach the number s(0); applying it once more, we reach s(s(0)); then s(s(s(0))), and so on through all the numbers. In the model A, if we start off at A_0 and repeatedly apply the function A_{s}, we get to 1 then 2 then 3, and so on. In fact, we'll reach — eventually — every element in the carrier set of A, which is why we can call this model 'reachable'. However, if we do this in the model B, starting at B_0, and repeatedly applying B_{s}, we'll never reach "junk". This explains the terminology 'reachable', but we'll stick with the more intuitive terminology of 'no junk'.

Exercise 5.28 *Let's construct another model of* ARITHMETIC *that has no junk, and let's call this model C. For the carrier set of C, we'll use unary numerals; i.e., the same as the carrier set of the term algebra of* UNARY*:*

$$C_{\mathsf{Nat}} = \{0, \mathsf{s}(0), \mathsf{s}(\mathsf{s}(0)), \ldots\} \ .$$

The implementations of 0 and s are straightforward: $C_0 = 0$, and $C_{\mathsf{s}}(x) = \mathsf{s}(x)$; i.e., C_{s} takes a unary numeral x and returns the unary numeral s(x).

Now define C_+. (Hint: use your answer to Exercise 2.13.) To complete the definition of C, we need to define C_*: do this.

The Lazy-programmer Model is another example of a model that has no junk. This example is particularly interesting because it works for *any* specification: for any equational theory (Σ, E), L_Σ is a Σ-model that satisfies all the equations in E (so it's a (Σ, E)-model), and it's also a model with no junk.

Exercise 5.29 *Check this.*

The Lazy-programmer Model is not 'too big', the problem is that it's clearly *too small*: there's only one element in the model, and that one element does all the work of representing all the data values of the ADT, even if those data values are meant to be different.

But what do we mean by 'meant to be different'? If we write a spec with equations, those equations tell us which terms are meant to be the same. For example, the equations in ARITHMETIC say that the terms s(s(0)) + s(0), s(s(s(0))) + 0, s(0) + s(s(0)), and s(s(s(0))) are all meant to be the same. But if the equations don't imply that two terms are meant to be the same, it's reasonable to assume that they're meant to be different For example, the equations in ARITHMETIC don't entail that 0 and s(0) are the same, so let's agree that this means they're meant to be different. You can put this down to incalcitrance, logical inertia, or an inverse version of Occam's Razor (don't unnecessarily introduce distinctions). In fact, it's a very pragmatic approach, and means that the notion of equality induced by a set of equations is *inductively* defined: the relation of equality is the smallest relation that relates everything that has to be related by the equations — just those things, and nothing else.

Let's try to capture the idea that an ideal model is not too small by defining *no confusion* to mean that a model doesn't have one element representing *different* terms, where 'different terms' means two terms that aren't forced to be equal by the equations in E.

Definition 5.23 *Let (Σ, E) be an equational theory. A (Σ, E)-model I has **no confusion** iff for all terms $t_1, t_2 \in (T_\Sigma)_s$, if $h(t_1) = h(t_2)$, then $t_1 \overset{*}{\leftrightarrow}_E t_2$.*

Looking back at the four models of ARITHMETIC on page 70, it should be clear that model A has no confusion, while all the others do have confusion: for example, in each of these models the terms 0 and s(s(0)) are represented by the same elements (it doesn't often happen that we're interested in proving that a particular model has no confusion, so we'll just stick with 'it should be clear'[5].

Let's summarise these requirements on a not-too-big-and-not-too-small model by calling an ideal model *initial*:

Definition 5.24 *Let (Σ, E) be an equational theory. A (Σ, E)-model I is **initial** iff*

- *I **has no junk**: for every $a \in I_s$, there is a term $t \in (T_\Sigma)_s$ such that $a = h_s(t)$, and*
- *I **has no confusion**: for all terms $t_1, t_2 \in (T_\Sigma)_s$, if $h(t_1) = h(t_2)$, then $t_1 \overset{*}{\leftrightarrow}_E t_2$.*

'No junk' means that every element a in the model represents some term t; 'no confusion' means that if two terms t_1 and t_2 are represented by the same element, then those terms must be equivalent.

As an example, model A (see page 70) of ARITHMETIC is initial ('obviously'!).

You're probably asking yourself: does every Maude spec have an initial model? The answer is: yes.

We saw that the term algebra is an initial model of any signature (and recall that a signature is just a Maude spec with no equations). For equational theories, we can also use the term algebra to get an initial model, but we have to *pretend* that equivalent terms are actually equal.

Think about how many different ways we can write the number zero using the syntax of ARITHMETIC. Here are a few of the infinitely many possibilities:

$$0, \quad 0 + 0, \quad 0 + 0 + 0, \quad 0 * 0, \quad 0 * s(0), \quad s(s(0)) * 0 \quad \text{and} \quad (0 + 0) * s(s(s(0))) \ .$$

Let's write $[0]_E$ for the set of all the possible ways of writing the number zero in ARITHMETIC. So

$$[0]_E \quad = \quad \{\, t \in (T_\Sigma)_{Nat} \mid t \overset{*}{\leftrightarrow}_E 0 \,\}$$

(where Σ is the signature of, and E the equations in, ARITHMETIC). Similarly,

$$s(0), \quad s(0) + 0, \quad 0 + s(0), \quad s(0) * s(0), \quad \text{and} \quad s(0) + (s(0) * 0)$$

[5]The computer science literature often has phrases like 'obviously' or 'it is left as an exercise for the reader' in cases where a proof is more difficult or lengthy than it is useful.

are all ways of writing down the number one; again, there are infinitely many possible ways, and let's write $[\,\mathsf{s}(\mathsf{0})\,]_E$ for the set of all the terms that represent the number one. We could go on, but let's generalise; let's write $[\,t\,]_E$ for the set of all the terms that are equivalent to t. And by the way, we're also generalising to any signature Σ and set E of Σ-equations, so

$$[\,t\,]_E \;=\; \{\, t' \mid t \overset{*}{\leftrightarrow}_E t' \,\}$$

for any equational theory (Σ, E). This transforms our 'equivalence' relation $\overset{*}{\leftrightarrow}_E$ into equality:

Proposition 5.2 *For all Σ-terms t and t', $t \overset{*}{\leftrightarrow}_E t'$ iff $[\,t\,]_E = [\,t'\,]_E$.*

The proof is left as an exercise for the reader.

We can now give a general recipe for constructing a model of any equational theory.

Definition 5.25 *Let (Σ, E) be an equational theory, with $\Sigma = (S, O)$. The **quotient term algebra**, T_Σ/E, is defined as follows.*

- *For each sort name $s \in S$, $\quad (T_\Sigma/E)_s \;=\; \{\, [\,t\,]_E \mid t \in (T_\Sigma)_s \,\}$*
- *for each constant $c \in O_{[],\,s}$, $\quad (T_\Sigma/E)_c \;=\; [\,c\,]_E$*
- *for $f \in O_{w,\,s}$, with $w = s_1 \cdots s_n$, and $t_i \in (T_\Sigma)_{s_i}$ (for $i = 1..n$)*

$$(T_\Sigma/E)_f([\,t_1\,]_E, \ldots, [\,t_n\,]_E) \;=\; [\,f(t_1, \ldots, t_n)\,]_E \;. \tag{5.16}$$

Proposition 5.3 *T_Σ/E is a (Σ, E)-model.*

Proposition 5.4 *T_Σ/E is an initial (Σ, E)-model.*

fmod and th; examples of ths (graphs, automata); protecting, including, and extending

Chapter 6

Program Verification

6.1 Specification

6.2 Implementation

6.3 Verification

Interlude. Turning things round (isn't necessarily smart).
