

# Chapter 1

## Maude

### 1.1 Abstract Data Types

An Abstract Data Type (ADT) consists of:

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

### 1.2 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.

<i>X</i>	<i>Y</i>	<i>X and Y</i>
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

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 :  $\rightarrow$  Bool .
  op not : Bool  $\rightarrow$  Bool .
  op _ and _ : Bool Bool  $\rightarrow$  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 (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.

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 .

```

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  $\langle 2, 26 \rangle$ . Then we might have a Maude specification containing the following declarations:

```

sort Pair .
op  ⟨ - , - ⟩ : Int Int → Pair .

```

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  $\rightarrow$  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 putting 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 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 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 Booleans and returns a Bool, this explains why Example 1(2) is a well-formed term of sort Bool.

In a sense, terms are like problems, just like '2  $\times$  6' is a problem whose answer is '12'. The term 'true and false' can be simplified, according to the truth table on page 1, 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 1** *Before we look at equations, have one last look at Example 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 :  $\rightarrow$  Bool .
  op not : Bool  $\rightarrow$  Bool .
  op _ and _ : Bool Bool  $\rightarrow$  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`:

<i>X</i>	<i>notX</i>
<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>

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 1. 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

## 1.3 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 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 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 mean 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<sup>1</sup>:

```

fmod BOOLEANS is

  sort Bool .

  ops true false : -> 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

```

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

```

<sup>1</sup>The file also includes the command `set include BOOL off`. This stops the interpreter looking at Maude’s ‘built-in’ specification of the Booleans, which uses the same notation.

```
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 2** Use Maude to check your answers to Exercise 1.

## 1.4 More Examples

### 1.4.1 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 3** Write out truth tables for the terms  $A$  or  $B$  and  $\text{not}(\text{not}(A) \text{ and } \text{not}(B))$  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** 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*

<i>A</i>	<i>B</i>	<i>A xor B</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>

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 :  $\rightarrow$  Bool .
  op not : Bool  $\rightarrow$  Bool .
  op _ and _ : Bool Bool  $\rightarrow$  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  $\rightarrow$  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 and B = not(B) .
eq false and 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 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  true | Y = false .
  eq  false | true = false .
  eq  false | false = true .

endfm

```

**Exercise 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  $\text{not}(A \text{ and } B)$ , 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.

### 1.4.2 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(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(0))))).
```

then the result should just be `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 — well just make sure that we get things right!)

**Exercise 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 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

$$s(s(0)) + s(s(0)) = s(0) + s(s(s(0)))$$

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

$$s(s(0)) + s(s(0)) = s(0) + s(s(s(0))) = 0 + s(s(s(s(0))))$$

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

$$s(s(0)) + s(s(0)) = s(0) + s(s(s(0))) = 0 + s(s(s(s(0)))) = s(s(s(s(0))))$$

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

$$s(s(s(s(0))))$$

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:

$$s(s(0)) + s(s(0)) = s( s(0) + s(s(0)) ) = s(s( 0 + s(s(0)) )) = s(s(s(s(0))))$$

**Exercise 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.

### 1.4.3 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  $\lambda$ -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 [] :  $\rightarrow$  List [ ctor ] .
  op _ : _ : Int List  $\rightarrow$  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  $\rightarrow$  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,

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

**Exercise 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 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. (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 `l : L`? Well, if `L` is empty, then `l` is the last element, so the result should be the empty list:

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

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 l J : Int .
var  L : List .
eq  removeLast(l : J : L) = l : 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(l : L) = l : 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 l : Int .
var L : List .
eq  removeLast([]) = [] .
```

```

ceq removeLast(l : L) = L if L == [].
ceq removeLast(l : L) = l : 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.)

#### 1.4.4 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

```

#### 1.4.5 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 l : Int .
  var S : Stack .
  eq top(push(l, S)) = l .
  eq pop(push(l, S)) = S .

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  $\rightarrow$  Int .  
op pop : NEStack  $\rightarrow$  Stack .  
  
var I : Int .  
var S : Stack .  
eq top(push(I, S)) = I .  
eq pop(push(I, S)) = S .
```