

# Chapter 2: Basic Maude

In this chapter, we will introduce the fundamental syntax of Maude, concentrating on those aspects of most concern to us. You can find manuals and tutorials on the [Maude website](#), and many of our examples in this chapter have been adapted from them (or in some cases, just "lifted" as-is). You are of course welcome to look at the material on the Maude site: however, for the most part it goes beyond what is required here, and may be rather difficult to interpret.

## 2.1. A Simple Maude Module

We start with a simple Maude module for the *Natural numbers*:

```
fmod BASIC-NAT is
  sort Nat .

  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm
```

This module introduces a new sort called `Nat`, and three operators. The first operator `op 0 : -> Nat .` is a function of no arguments, and is hence a *constant*, since (mathematically) any function which takes no arguments must always return the same value. The second operator `op s : Nat -> Nat .` is the successor function (or more accurately, *is intended to represent* the successor function). The third operator `op _+_ : Nat Nat -> Nat .` is the addition function: notice we use *mixfix* notation for addition, where arguments are intended to replace the underscores `_`. We do this for notational convenience, and it is our responsibility to ensure the number of underscores matches the number of arguments.

Following the operators, we define the behaviour of the addition operator by *structural induction* in the usual way. Notice that we declare all the variables that appear in the equations. There are no equations defining the behaviour of `s` and `0`. Strictly speaking, these *generate* data and are called *constructors*. In fact, `s` and `0` between them can generate all the possible elements of sort `Nat`, and the addition operator just "chooses" one of them: the result of an addition will either be `0` or of the form `s(s(...s(0)...))`. Constructors do not need defining equations, though we can if we wish denote them in our algebras by `[ctor]`: for example,

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

This does not actually affect the logic, and we will generally omit it. Incidentally, there are a number of other things that can be added to operators like this, and we will see some later.

### 2.1.1. A Note on Whitespace

Maude has very liberal views on identifiers which, combined with mixfix notation, provides a flexible syntax, that is very convenient. However, it has consequences - the most obvious to new users being its "sensitivity" to whitespace. Notice in the example above that we have put spaces around all the key tokens - `:`, `->` - even before the terminal `.` ending each statement. A "normal" programming language would not require this - but Maude does. Failing to observe this is the most common syntactic problem new users have in my experience.

### 2.1.2. Another Example - Booleans

Here is another *simple example*, that we introduce largely because we will need it later. Notice the use of `[ctor]` here.

```
fmod BOOLEAN is
  sort Boolean .

  op true : -> Boolean [ctor] .
  op false : -> Boolean [ctor] .
```

```

op not : Boolean -> Boolean .
op _and_ : Boolean Boolean -> Boolean .
op _or_ : Boolean Boolean -> Boolean .

var A : Boolean .

eq not true = false .
eq not false = true .
eq true and A = A .
eq false and A = false .
eq true or A = true .
eq false or A = A .

endfm

```

Notice that we have made no provision in this example for the associativity and commutativity of operators `+`, `and` and `or`. We address this in a [later section](#). Also, Maude has a built-in version of the Booleans (the sort is `Bool`).

## 2.2. Module Hierarchies

To simplify and organise our descriptions of systems we can structure them hierarchically by a system of importation. Maude provides two ways of doing this: `protecting` and `importing`. Here is [an example](#) using `protecting`

```

fmod NAT+OPS is
  protecting BOOLEAN .
  protecting BASIC-NAT .

  ops *_ _- : Nat Nat -> Nat .
  ops <=_ _>_ : Nat Nat -> Bool .

  vars N M : Nat .

  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq 0 - N = 0 .
  eq s(M) - 0 = s(M) .
  eq s(M) - s(N) = M - N .
  eq 0 <= N = true .
  eq s(M) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
  eq M > N = not(M <= N) .

endfm

```

By using `protecting` we are saying that we are, essentially, not changing the behaviour of anything in the modules we are importing. However, in the next example (natural numbers modulo 3) we must use `importing` because we are modifying the behaviour of `BASIC-NAT`:

```

fmod NAT 3 is
  including BASIC-NAT .

  var N : Nat .

  eq s(s(s(N))) = N .

endfm

```

Generally, we will be using `protecting`.

## 2.3. Rewriting

We have seen how to create simple modules: how do we go about using them. [Download](#) the first example, and type

```
maude.linux basicnat.maude
```

You should see something like:

```

      \|||||/
    --- Welcome to Maude ---
      /|||||/
Maude version 1.0.5 built: Apr  5 2000 15:56:52
  Copyright 1997-2000 SRI International
      Thu Feb 14 14:17:01 2002

```

Maude>

To evaluate something you must *reduce* it, using the `reduce` command (shortened to `red`). At the prompt, type:

```
red s(0) + s(s(0)) .
```

(Notice the final ``." and the space before it!). You should see:

```

reduce in BASIC-NAT : s(0) + s(s(0)) .
rewrites: 2 in -10ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(0)))

```

You have just asked Maude to evaluate  $s(0)+s(s(0))$  (i.e.  $1+2$ ) and got the answer  $s(s(s(0)))$  (i.e. 3). To see things in a little more detail, type `set trace on .` before performing the reduction. You should see:

```

Maude> set trace on .
Maude> red s(0) + s(s(0)) .
reduce in BASIC-NAT : s(0) + s(s(0)) .
***** equation
eq s(M) + N = s(M + N) .
M:Nat --> 0
N:Nat --> s(s(0))
s(0) + s(s(0))
---->
s(0 + s(s(0)))
***** equation
eq 0 + N = N .
N:Nat --> s(s(0))
0 + s(s(0))
---->
s(s(0))
rewrites: 2 in -10ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(0)))

```

Maude first applies the equation  $s(M) + N = s(M + N)$  to rewrite  $s(0) + s(s(0)) = s(s(0) + s(s(0)))$ , and then applies equation  $0 + N = N$  to rewrite  $0 + s(s(0))$  to  $s(s(0))$ . Try some experiments with BASIC-NAT, BOOLEAN and NAT+OPS to get the hang of things. You'll probably want to do `set trace off .` (or restart Maude) as your expressions get more complex, or you will find the output gets a bit ``voluminous". By the way, to exit Maude type `quit`.

### 2.3.1. Small Technical Point

As you experiment with BOOLEAN you will probably get some complaints from Maude about ``multiple distinct parses". This is because there is already a built in version of the Booleans (`Bool`) with constants `true` and `false`. Unless you distinguish them, it cannot tell them apart. There are two ways to do this. First, rename `true` and `false` to (e.g.) `tt` and `ff` (try it), or secondly to *explicitly* tell Maude the sort(s) you intend it to use. For example, instead of typing

```
red true and true .
```

type

```
red (true).Boolean and true .
```

You could have typed `(true).Boolean` and `(true).Boolean`, but this is not necessary (why?).

## 2.4. Associativity and Commutivity

We have discussed associative and commutative operators in the term rewriting chapter, and clearly some of the

operators above are commutative and/or associative. We cannot just add equations asserting this without getting a non-terminating rewriting system, so instead we annotate the operators with the property using the `assoc` and `comm` keywords. For example:

```
op __ : Nat Nat -> Nat [assoc] .
op __+__ : Nat Nat -> Nat [assoc comm] .
```

Modify the modules in this chapter appropriately, and try some experiments with them.

## 2.5. Conditional Rewriting (and Other Things...)

Commonly we will only wish to apply rewrite rules if some condition is true. We can do this using *conditional equations*. Here is an example

```
fmod BITS is
  protecting MACHINE-INT .
  sorts Bit Bits .
  subsort Bit < Bits .

  ops 0 1 : -> Bit .

  op __ : Bits Bits -> Bits [assoc prec 1 gather (e E)] .
  .
  op bits : Bits MachineInt MachineInt -> Bits .

  vars S T : Bits .
  vars B C : Bit .
  var L : Bool .
  var I J : MachineInt .

  *** Extract Bits...
  eq bits(S B,0,0) = B .
  eq bits(B,J,0) = B .
  ceq bits(S B,J,0) = bits(S, J - 1,0) B if J > 0 .
  ceq bits(S B,J,I) = bits(S,J - 1,I - 1) if I > 0 and J > 0 .
end
```

This example is a cut-down version of the BITS module that will be used in microprocessor specifications for defining all sorts of "useful" operators on bit strings. This module introduces a number of new concepts

- Comments, denoted by `***`
- Subsorting: we have introduced two sorts `Bit` and `Bits`, and have stated that `Bit` is a subsort of `Bits`. This means that all operators that are defined over `Bits` (i.e. strings of bits of length greater than 1) are also defined over single bits (`Bit`). This will come in handy later on.
- The `__` operator (two underscores) with no name: just write two bits/bit strings next to each other. This is a constructor (the concatenation operator).
- Conditional Equations denoted by `ceq`.
- `prec 1 gather (e E)` which controls the construction of parse trees.

The last point is complex, and I don't want to go into it much as it will not concern us. If we don't include `prec 1 gather (e E)` and try the following rewrite:

```
maude> red 1 0 + 1 0 .
result Bits: 1 1 0
```

which is not the expected result of `1 0 0`. This is because the default parse is

```
1 (0 + 1) 0
```

and `0 + 1 = 1`. We could of course bracket the term to force the parse we want, but if we use `prec 1 gather (e E)` we don't have to (and we would sometimes forget).

We will use conditional equations extensively: they are illustrated above in the definition of the `bits` operator (which extracts a substring from a [longer] bitstring). The structure and use is (hopefully!) obvious. Two useful operators that are commonly used in conditional equations are `==`, which is *semantic equality* (i.e. equality of two terms after they have been reduced) and `≠` which is *semantic inequality*. Note a common error is to use

" instead of ``=".

Another conditional operator that can be used instead of/as well as conditional equations is `if_then_else_fi`. Here is an example of it, also taken from BITS

```

op _>_ : Bits Bits -> Bool [prec 4 gather (E E)] .

*** Greater than
eq 0 > S = false .
eq 1 > (0).Bit = true .
eq 1 > (1).Bit = false .
eq B > (0 S) = B > S .
eq B > (1 S) = false .
eq (1 S) > B = true .
eq (B S) > (C T)
  = if | %(B S) | > | %(C T) | then
      true
    else if | %(B S) | < | %(C T) | then
      false
    else
      (S > T)
    fi
  fi .

```