Maude 2.0 Primer

Version 1.0

Theodore McCombs

August 2003

Introduction

Hi! Welcome to the Maude primer. If you're looking for an in-depth exploration of the capabilities and mathematical foundation of Maude 2.0, this is not it. You're looking for the Maude manual, a wonderful comprehensive tome that weighs about as much as a newborn child. This primer is meant as a more casual, more cursory introduction to Maude; the main goal is to get the beginning user, whether he/she have an extensive or just a basic background in mathematics and modeling, programming in Maude as quickly and painlessly as possible.

So, what is Maude?

Maude is a programming language that models systems and the actions within those systems. The rest of the primer will be spent clarifying and illustrating this statement.

Maude is *powerful*. It can model almost anything, from the set of rational numbers to a biological system to the programming language Maude itself. Anything you can write about, talk about, or describe with human language, you can express in Maude code.

Maude is *simple*. Its semantics is based on the fundamentals of category theory, which is pretty intuitive and straightforward until a mathematician tries to describe it formally with symbols and Greek letters. While such notation is essential to the detailed understanding of Maude, the beginning user does not need to know it to start programming; therefore, this tutorial will keep it simple. Compared to most languages, Maude does not have much syntax to memorize, only a small set of key words and symbols, and some general guidelines and conventions to keep track of.

Maude is *concrete*. Or, depending on how you look at it, it is extremely abstract. Its design allows such flexibility that the syntax may seem rather abstract, and alone, it is. However, this means that, as the user programs, he/she is very close to the problem itself. To put it another way, the majority of programming languages distance the programmer from the model with a gaggle of special keywords and syntax whose design and protocol is more or less hidden from the user. In a Maude model, pretty much every object and artifact is open to the programmer's scrutiny and whim, and the majority of the model has probably been actually written by the user. So you need not worry about how to declare an `int` or a `float` correctly – just look in the modules `INT` and `FLOAT` for the code of the model.

Maude is *challenging*. This is not to say it is difficult, or complex, but rather, that the ingenuity and mindset required for a certain algorithm are often strikingly original. It challenges the programmer to be clever; instead of solving a problem by bashing it with enough global variables and functions, one is asked to write streamlined, efficient code, which, often enough, is a thrill in itself.

This primer is designed as an overview of the most important and powerful features that make Maude unique, and most importantly, how to use them. Short quizzes follow each chapter to

drive home the lessons; they are not especially tricky, and I urge the reader to try his or her hand at them. Finally, I cannot repeat enough (this will certainly not be the last time) that the primer was never intended to be completely comprehensive, and if you don't find what you're looking for here, there's a host of documentation and theory in the Maude manual or the plethora of papers written by the Maude team.

To download Maude 2.0, go to http://maude.cs.uiuc.edu and you can find versions of Maude for several platforms, the Maude sources, papers on Maude, the Manual, and contact information. Good luck, and enjoy!

1. FOUNDATION

1.1 Modules

The module is the key concept of Maude. It is essentially a set of definitions. These define a collection of operations and how they interact, or, mathematically speaking, they define an *algebra*. An algebra is a set of sets and the operations on them. In Maude, a module will provide a collection of sorts and a collection of operations on these sorts, as well as the information necessary to reduce and rewrite expressions that the user inputs into the Maude environment.

There are three types of modules in Maude: the *functional module* (`fmod`), *the system module* (`mod`), and the *object-oriented module* (`omod`). The differences between these will appear as we look at the capabilities and specifics of each one. Each module is declared with the key terms:

```
fmod NAME is … endfm
mod NAME is … endm
omod NAME is … endom
```

where the ellipses denote all the declarations and statements in between the beginning of the module and the end of it. Whitespace does not make a difference in Maude.

There are actually two separate levels of Maude: Core Maude and Full Maude. Anything in Core Maude is in Full Maude, but not vice versa. System modules and functional modules are both in Core Maude, but not object-oriented modules. To use Full Maude, all commands and modules must be entered in enclosed in parentheses, so, in effect, the declaration for any object-oriented module will look like:

```
(omod NAME is … endom)
```

As in most programming languages, one can import a module from another. This means that we can include all the declarations and definitions that another module made in a new one without being redundant. One uses the key words `protecting`, `extending`, or `including`, depending on how we use the imported module. "`protecting`" essentially means that the declarations in the imported module are not modified in any way, that all the defined operations and sorts are used strictly as they are in the imported module. "`including`" means that one can change the sense in which the declarations were used. For example, if you wanted import the module `INT` (the provided module for integers) but wanted to define its addition operator to multiply instead of add, you would use `including`. The syntax is simply:

```
protecting MODULENAME .
```

or

```
including MODULENAME .
```

or

```
extending MODULENAME .
```

The `extending` importation falls somewhere between these two extremes, but a deeper knowledge of modules is needed to explain – we will return to it in the next chapter.

Don't forget the period after the importation. All statements like these, including all declarations, need the period. Only module definitions don't need the period.

1.2 Sorts and Variables

`***`Before continuing, it would be beneficial to point out that Maude supports comments.

`---`Three asterisks or three dashes in a row set apart the rest of the line as a comment.

A *sort* is a category for values. For example, the number one could be a value of sort "`integer`," or perhaps just "`number`," or even "`PlaceInLine`." A sort can pretty much describe any type of value, including lists and stacks of other values. Sort names should be specific, increasing readability and the tightness of the code. Sort names, like all other identifiers in Maude, may not contain whitespace, a comma, or any of the three bracket types: '`(`', '`[`', and '`{`' unless immediately preceded by an apostrophe, that is, backquote: '``'. Additionally, sort names in particular may not contain periods or colons – as we will see later, the reason is that these special symbols are heavily involved in the syntax of Maude, and using them incorrectly will set off all kinds of fireworks.

A sort is declared within the module, with the key word `sort` and a period at the end.

```
sort integer .
sorts integer decimal .
```

There are also *subsorts*, which are further specific groups all belonging to the same sort. For example, the sort of real numbers `real` could have `irrational` and `rational` as subsorts, and `rational` could have `integer` and `fraction`, `integer` could have `positive` and `negative`, etc. etc. One declares a subsort by first declaring the sorts, and then situating one sort within another with the key word `subsort` (or `subsorts`) and the less-than sign (<).

```
sorts Real Irrational Rational Integer Fraction Positive Negative .
subsorts Irrational Rational < Real .
subsort Fraction < Rational .
subsorts Positive Negative < Integer < Rational .
```

All this means is that both `irrational` and `rational` are subsorts of `real`, that `fraction` is a subsort of `real`, and that `positive` and `negative` are subsorts of `integer` which is a subsort of `rational`.

A *variable* is an indefinite value for a sort. Just as *x* is a common variable for a number, one can declare a variable `x` as being of sort `number`. Or, one could declare `c1`, `c2`, and `c3` as variables

of sort `color`. Variables are never used as constants; one should not declare `red` as a variable of sort `color`. Unlike variables in C++ and Java and other common programming languages, Maude variables never have a definite value assigned to them. They cannot carry from one operation to another. The only important use of variables is as placeholders, when defining operations through equations and rewrite laws, which will come later.

Variables are declared with the key words `var` or `vars`, followed by the name(s) of the variables, then a colon, and then the sort to which each variable belongs, and finally a period. Note that if one declares multiple variables using `vars`, all the variables must belong to the same sort.

```
var x : number .
vars c1 c2 c3 : color .
```

The expression `var x : [number] .` declares a variable for the *kind* of number as opposed to the *sort*. We will discuss kinds later on in the chapter, but suffice to point out that variables can be declared as such.

1.3 Operations

As mentioned earlier, describing an algebra (one of the key mathematical uses of Maude) consists of declaring sorts and declaring the operations that act on these sorts. An operation can be thought of as a pathway between sorts. For example, if one had a sorts `x-coordinate`, `y-coordinate`, and `point`, one could create an operator "`( _ , _ )`" to combine an x-coordinate and a y-coordinate into a point on a Cartesian plane. Or, one could create an operator "`bake`" to connect sorts "`batter`" and "`cake`." Surprisingly, the activities of operators will become clearer as we look at the syntax behind them.

Maude understands both prefix and mixfix notation for operations. Prefix notation is used most widely, in major programming languages like C++ and Java, and indeed the notation "$f(x)$" is prefix. Prefix notation consists of the name of the operator, followed by its arguments in parentheses and separated by commas. To call the `+` operator on variables `x` and `y` using prefix notation, one would write "`+(x, y)`." As one can see, this notation makes more sense for operations like "`bounce(ball3)`."

The alternative is mixfix notation, which, unlike prefix in Maude, is declared with specific places for the variables. While one would declare the addition operator in prefix notation as "`op +`," in mixfix it would be "`op _+_`." Or, if for some reason you wanted an operation to add the sum of two numbers to a list, prefix notation would declare it "`op AddSumToList`" and call it with "`AddSumToList(x, y, List1)`"; mixfix would declare the operation "`op AddSum_+_to_`" and call it as "`AddSum x + y to List1`."

One declares an operation in either prefix or mixfix using the key word `op`, followed by the name of the operation, then a colon, then the name(s) of the sort(s) fed into the operation, then an arrow (`->`), the name of the sort of the result of the operation, and finally a period. Both the prefix and mixfix declarations are shown for the declaration of an addition operator adding two natural numbers (sort `Nat`).

```
op + : Nat Nat -> Nat .
op _+_ : Nat Nat -> Nat .
```

Similarly, one can declare two operations with the same sort arguments and sort results by using the key word `ops`.

```
ops +  * : Nat Nat -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

Both the addition and multiplication operations connect two variables of sort `Nat`, and return a variable of sort `Nat`.

Operation names may contain the special characters, such as the parentheses and brackets, commas, and periods. Maude parses them with backquotes, but you don't have to worry about that. But if you do use parentheses in an operator name, you have to declare it with parentheses around that name. You may declare two operators with the same name – this is called *operator overloading* – but it is a powerful tool that should be handled carefully. We will further discuss operator overloading in Chapter 2.

1.4 Constructors and Operator Attributes

There is a special type of operation supported in Maude called a constructor. To designate a function as a constructor, one adds "`[ctor]`" after the result sort name and before the period. A constructor is a fundamental operation that defines the basics of the algebra. We won't be able to fully appreciate this definition until the next chapter, but let's give it a go anyways.

One approach to understanding the role of constructors is via the constant operation, which acts as the equivalent of a constant in other programming languages. But instead of a variable (remember that Maude variables do not carry or store values), Maude constants are functions with no independent variable, in the sense that "2" can be represented by the constant mathematical function $f(x) = 2$. Constants leave the place for the argument sorts blank.

```
op pi : -> Irrational .
ops red blue yellow : -> Color .
```

Of course, one still needs to designate the result sort to which the constant belongs. What does this have to do with a constructor? Constants will often be constructors, because they often define a basic aspect of the algebra. For example, if we were to try and model the trigonometric functions in a module, we'd certainly need the constant `pi`, which we use to express angles in radians. Without `pi`, we're helpless. In this case, we therefore set `pi` as a constructor:

```
op pi : -> Irrational [ctor] .
```

On the other hand, if we simply want to define the set of irrational numbers, the constant `pi` is superfluous, not exactly a waste of space, but simply an extra feature that isn't necessary to understand irrational numbers. In this case, we do not make `pi` a constructor.

It's kind of tricky to decide what operations should be constructors, but only experience remedies this. One excellent example can be found in the Peano notation for natural numbers. Peano proved that the set of natural numbers could all be described with a constant 0 and a successor operator s( ) that took a natural number and returned the natural number after it. For example, two could be expressed as the s(s(0)). Every other closed operation on the natural numbers can be defined with the combination of these fundamental operators, and so we set these as constructors. On the other hand, an addition operator isn't fundamental to the Peano notation; it could support one, but does not necessitate it. Therefore, the addition operator is not a constructor.

The Peano notation of natural numbers is given below:

```
fmod PEANO-NAT is
      sort Nat .
      op 0 : -> Nat [ctor] .
      op s : Nat -> Nat [ctor] .
endfm
```

One will note that that the operator `s` is not declared as `s_`, so it must be in prefix notation. And, it should be obvious that one need not worry about prefix vs. mixfix for constant functions, because they take no arguments!

Another common and good-to-know example is the basic List algebra. A bare-minimum list needs a sort `List` and an element sort, or as commonly represented, `Elt`, a constant for empty lists, and a concatenation operator. It can support, but does not need, operations to return the head or tail of list the size of the list, etc. The module below represents the bare minimum. Note that, because the concatenation operator is necessary to define the bare-minimum, it is a constructor.

```
fmod BASIC-LIST is
   sorts List Elt .
   subsort Elt < List .
   op nil : -> List [ctor] .
   op __ : List List -> List [ctor] .
   vars E1 E2 : Elt .    vars L1 L2 : List .
endfm
```

There are a few important things to note in this example. The concatenation operator is the double underscore. This operation is in mixfix notation, and would be called with variables `L1` and `L2` as "`L1 L2`." `Elt` is declared as a subsort of `List`, because an element of a list is really just a special case of a list: a list of size one (called a singleton). This also means we can form a list by concatenating two elements, because an operator that takes List will also take a subsort of List. Otherwise, we would have to create an operator that transforms an element into a list, and this is much easier. This way, the Maude compiler understands not only "`L1 L2`" but also "`E1 E2`" and "`E1 L1`." And finally, the constant constructor `nil` is a common way to name a list of size zero.

A list is a perfect example for illustrating the use of the three key flags `assoc`, `comm`, and `id`, which impose the *equational attributes* of associativity, commutativity, and identity on any binary operator (whether it is a constructor or not). Associativity just means that the list formed by "`(L1 L2) L3`" is the same as the list formed by "`L1 (L2 L3)`" is the same as the list formed by "`L1 L2 L3`." Associativity is declared by adding the key word `assoc` after `ctor` in the brackets.

```
op __ : List List -> List [ctor assoc] .
```

Identity is an extremely important attribute for lists. The identity property of addition is "n + 0 = n." Similarly, the identity property of lists is that "`L1 nil`" is the same as just "`L1`." In other words, concatenating a list with an empty list reduces to the original list. Identity is declared with the key word `id:` followed by the null constant that fulfills the identity property, in this case `nil`. One may also define identities for the left and right arguments (remember, all these flags only apply to binary operators: that is, operators with two arguments), using the flags `left id:` and `right id:`. This doesn't make much sense for a concatenation operator, but consider the difference between the identity declarations `jack kills nobody` and `nobody kills jack`. Illustrated below is the general identity flag:

```
op __ : List List -> List [ctor assoc id: nil] .
```

Commutativity is a property that technically does not belong to lists, but rather to sets (or rather, to 'multi-sets,' but that's not really the issue here). Commutativity means that order does not matter, that "`S1 S2 S3`" is the same as "`S1 S3 S2`" is the same as "`S2 S1 S3`" etc. The order of the equational attribute flags within the brackets does not matter.

```
op __ : Set Set -> Set [ctor assoc comm id: none] .
```

There is still another flag to mention, `idem`, which denotes the equational attribute idempotency, the property that repeated elements are discarded. Though an important property of sets, we will suffice with the mention of idempotency, since as of yet the `assoc` and `idem` flags may not be used together in Maude.

A useful flag for unary operators (operators only taking one argument) that must be iterated over and over again – say, for example, the `s_` operator, where 'five' takes the monstrous form of `s s s s s 0` – is the `iter` flag, which allows such chains of iteration to be expressed as a single instance of the operator, raised to the number of iterations. To whit, we may write `s s s s s 0` as `s_^5(0)`. This notation may be used for both input and output. The flag `iter`, like the other flags, falls between the brackets following the operator declaration.

1.5 Kinds

It may have occurred to you already that an improperly used operator can make a sort go... wrong. For an example, let's consider a canine pedigree chart that maps the genealogy of a particular puppy:

```
sort Dog .
ops bloodhound collie dalmatian pitbull schnauzer : -> Dog .
op breed : Dog Dog -> Dog [ctor] .
```

So a mutt with a little Dalmatian, Pitbull, and Schnauzer might have the pedigree chart `breed(breed(pitbull, breed(pitbull, dalmatian)), schnauzer)`. But what poor creature results from the expression `breed(schnauzer, penguin)`?

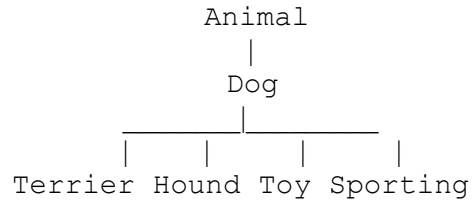
```
sort Animal .
subsort Dog < Animal .
op penguin : -> Animal .
```

The expression `breed(schnauzer, penguin)` is an *error term*, and while it is certainly not of the sort `Dog` or even `Animal`, it is of the *kind* `[Animal]`. Whenever we define a sort, Maude automatically defines a corresponding *kind*, a 'supersort' that groups together all other sorts connected to that sort (I will explain this soon) and all 'interfamilial' errors.

When we declare a sort hierarchy using `subsort`, we define a *connected component* for Maude. If, say, we write

```
sorts Terrier Hound Toy Sporting .
subsorts Terrier Hound Toy Sporting < Dog .
```

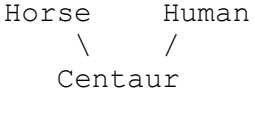
we create a graph of connected sorts. Our current graph looks something like:



This whole entire graph is connected component. If we were to declare another sort `Vegetable` and not declare any subsort relations with any sort on the above graph, `Vegetable` would not be part of the connected component. `Vegetable` would be its own, separate, singleton connected component.

A kind may be defined by any sort in the connected component. Thus, a breed between a Jack-Russel-terrier (a member of `Terrier`) and a frog will be a member of `[Animal]`, or `[Dog]` or `[Terrier]`, though Maude will always refer to it as `[Animal]`. If there be more than one topmost sort, as in the example below:

```
sorts Human Horse Centaur .

subsort Centaur < Human.

subsort Centaur < Horse .
```



then the kind may also be written as `[Horse,Human]`, which is how Maude understands the kind.

Moreover, the only error terms included in a kind are those that involve members of the connected component. Breeding a collie and a bromeliad will *not* be a member of `[Animal]` or `[Vegetable]`; the term `breed(collie, bromeliad)` will simply not parse, i.e., Maude will return a warning and do nothing with the collie-flower.

But are kinds just kindnesses towards malformed hybrids, or do they actually have some real use to them? Well, error-handling is extremely important to a computer language, so consider it a kindness towards Maude and the Maude user. In the next chapter, we'll also see that kinds can be handled by operations and equations in the same way that sorts can, with the result that certain error expressions can be simplified into non-errors – in other words, we can breed out the penguins in our dogs.

1.6 Common Supplied Modules: `NAT`, `QID`, and `STRING`

Maude thankfully has a library (albeit small) of common modules to import if needed. `Nat`s are an alternative to defining your own `Nat` or `Integer` or `Num` sort: the module `NAT` includes sort `Nat`, addition `_+_`, symmetric difference `sd`, multiplication `_*_`, quotient `_quo_`, modulo `_rem_`, and a host of other useful operations, and all the natural numbers written in normal, numeric notation. One can simply say "`2 + 16`" rather than "`s s 0 + s s s …` well, you get the point. However, the `NAT` module supports the Peano notation as well. The provided

modules `INT`, `FLOAT`, and `RAT` support integer, floating point, and rational numbers, respectively.

Another extremely useful standard module is `QID`, or Quoted IDentifier. An apostrophe just before any word creates a quoted identifier, which can be used as a name. For example, if we wanted to create a sort `Name` and didn't want to define a whole slew of constants we could write:

```
fmod NAME is
   protecting QID .
   sort Name .
   subsort Qid < Name .
endfm
```

Then, `'Ted` would be a term of sort `Name`, without having to define constant constructor.

The standard library also provides a `STRING` module, while handles strings of characters and provides useful functions for searching and determining length. A `String` is any group of characters enclosed in quote marks, for example, `"hello world!"` Strings can be concatenated with the _+_ operator, so `"hello" + "world"` is equal to `"helloworld"`. The characters of a string are indexed by natural numbers, starting at the beginning with zero. The `substr` function returns a part of the string with a certain starting index and length; `substr("hello world!", 2, 3)` would return `"llo"`, for example. The `find` function searches for a substring, starting at a certain index and going forward; the `rfind` function searches backwards from the given index.

```
find("hello world!", "orl", 0) = find("hello world!", "orl", 4) = 7
rfind("hello world!", "orl", 0) = notFound
rfind("hello world!", "orl", 10) = 7
```

This summary does not pretend to be exhaustive in any way. To truly appreciate the power of the standard library, please refer to the Maude 2.0 manual (available at the website), Part I, Chapter 7.

Sorts and operations are the core foundation of Maude. In the next few chapters, we'll look at how each type of module creates algebras out of this foundation.

EXERCISES FOR CHAPTER 1

1. The three types of modules in Maude are declared as fmod (functional ..), mod (system mod), and omod (o-o mod). They syntactically stand for _____ module, _____ module, and _____ module, respectively. The only type of module that is not in Core Maude, but is in Full Maude, is the _____.

2. Identify the following operations as prefix, mixfix, or constant:

```
op _^_ : Nat Nat -> Nat .                  _____
op smile : -> Expression .                 _____
op to_from_ : Name Name -> Valentine .     _____
op grab : Thing -> Action .                _____
```

3. When defining the division operator for the natural numbers, one must address the problem of dividing by zero. Do the following on the lines below. a) Define two sorts, `Nat` and `NzNat`, which stands for a non-zero natural number. b) Declare `NzNat` a subsort of `Nat`. c) Define a division operation that takes a `Nat` and an `NzNat`, and returns a `Nat`.

   a) _____

   b) _____

   c) _____

4. Say I want to make an operation that takes a person's *name* and *age*, and returns a statement such as "*Johnny* is *3* years-old".

   a) Choose two standard modules you think I should import for this operation:

   _____ and _____

   b) Fill in the blanks for the operation declaration with the correct standard sorts:

```
op _is_years-old : _____________ _____________
                                    -> AgeStatement [ctor] .
```

   c) Assume that this is the only operation that uses the two imported modules. How should I import them: using `including` or `protecting`? _____

5. Let us define two sorts, `Queue` and `Grab-bag`. In a `Queue`, the order of the elements matter, and so we want to exclude commutativity from the list of its properties. In a `Grab-bag`, order doesn't matter, so we want commutativity as a property. Furthermore, we want an identity property defined for both, with respect to the `nothing` constant. Associativity we definitely want for both.

   Below are two constructors, one for `Queue` and another for `Grab-bag`. Fill in the blanks between the brackets, putting in all the syntax necessary to fulfill the above requirements.

```
op __ : Queue Queue -> Queue   [_______________________________] .
op __ : Grab-bag Grab-bag -> Grab-bag
                               [_______________________________] .
```

6. The combined use of subsorts with operation declarations can be very powerful, and consequently rather tricky. By defining an operation for a sort, one defines it for all of its subsorts. One can think of the < symbol as short for "qualifies as a," e.g., `integer` *qualifies as a* `rational`. With this in mind, and given the following sort and operation declarations,

```
sorts Plea Inquiry Assertion Question Speech .
sort LogEntry .
subsorts Inquiry < Question < Plea .
subsorts Assertion Question < Speech .

op respond : Question -> Speech .
op record : Speech Speech -> LogEntry .

ops p1 p2 : -> Plea .        ops i1 i2 : -> Inquiry .
ops a1 a2 : -> Assertion .   ops q1 q2 : -> Question .
ops s1 s2 : -> Speech .
```

mark which expressions will parse correctly, i.e., which will follow the rules of the declarations above.

_____	`respond(q1)`	_____
_____	`record(q1, i1)`	_____
_____	`respond(record(q1, i1))`	_____
_____	`record(q1, respond(q1))`	_____
_____	`respond(p1)`	_____
_____	`record(p1, p2)`	_____
_____	`record(respond(i1), s1)`	_____
_____	`record(i1, respond(i1))`	_____
_____	`respond(s1)`	_____
_____	`respond(respond(q1))`	_____

In the second blanks, mark which sort or kind will be the result of the expression, or `no parse` for those terms that will simply not parse at all.

2. FUNCTIONAL MODULES

2.1 Equations

Equations are much easier to describe in form than in purpose. They follow the same structure, rules of order of operation, etc. of any normal mathematical equation. The syntax uses the key word `eq`, followed by two expressions separated by the key symbol `=`, and then a period.

Remember the Peano notation of natural numbers example? The module defined an algebra of the Peano notation, though not a very interesting one. If we add the addition operator, it becomes slightly more complicated, and we use equations to define this operator.

```
fmod PEANO-NAT-EXTRA is
   sort Nat .
   op 0 : -> Nat [ctor] .
   op s : Nat -> Nat [ctor iter] .
   op _+_ : Nat Nat -> Nat .
   vars M N : Nat .
   eq 0 + N = N .
   eq s(M) + N = s(M + N) .
endfm
```

The whole idea of equations is to provide the Maude interpreter with certain rules to simplify an expression. One rule of addition is that zero plus a number reduces to that same number. Without this equation, Maude would not be able to reduce `0 + s(s(s(0)))` to `s(s(s(0)))`. The second equation is a more general simplification rule for addition, which would help reduce `s(s(0)) + s(0) to s(s(0) + s(0))` and then `to s(s(0 + s(0)))`. The first equation would then reduce the expression to its most simple form, `s(s(s(0)))`.

This brings up another unique quality of constructors: expressions with only constructors cannot be simplified. There are no equations to define how to simplify constructors, and there shouldn't be. The ultimate goal of a set of equations in a module is to provide rules by which any expression involving the sorts and operations in the module can be simplified to the constructors of the algebra – this is called the *canonical form* of the expression.

The use of variables in equations is really the only use of variables in a module at all. As mentioned before, variables do not carry actual values. Rather, they stand for any instance of a certain sort. For example, when in the example above we used `M` and `N` as variables of type `Nat`, the expression `s(M) + N` really just means "take the successor of any `Nat`[ural number] and add it to another `Nat`." Variables are like the spaces in Mad-Libs: insert any instance of the sort in the blank, and the expression works.

Maude also supports *on-the-fly variable declaration*, and yes, it really is as fun as the name sounds. When, for example, our algebra forces us to write an equation with a variable we only use once, we have two options: waste space by declaring a variable in the usual manner, or

declare it on-the-fly, within the equation. If we have, say, a card catalog constructor, and an operator that retrieves the author name:

```
op n_t_pd_l_ : Name Title PublDate Location -> CatalogCard [ctor] .
op author : CatalogCard -> Name .
```

and we really don't have any other equations that involve instances of these sorts, we can, instead of declaring the variables outside the function, simply write:

```
eq author(n N:Name t T:Title pd P:PublDate l L:Location) = N:Name .
```

Note that on-the-fly variables include no whitespace between the variable name, the colon, and the variable sort. The scope of such variables is the equation it's declared in; outside this equation it doesn't exist.

Now, there are several common strategies or guidelines for writing equations that simplify well. The first and most important strategy is recursion. Nearly every set of equations is defined with some level of recursion in mind. If you don't like recursion, Maude is definitely not the programming language for you. Take another look at the two equations for the addition operator.

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

The second equation employs recursion by calling the operator _+_ again on the right hand side, and the first equation is the case that ends the recursion. A more obvious example of recursive definition appears in multiplication:

```
fmod PEANO-NAT-MULT is
   protecting PEANO-NAT-EXTRA .
   op _*_ : Nat Nat -> Nat .
   vars M N : Nat .
   eq N * 0 = 0 .
   eq N * s(M) = N + (N * M) .
endfm
```

If one were to try 3 * 5 (I've dropped the successor notation for simplicity's sake, but one can imagine all the successors of successors etc.), the recursion would look something like

$$
\begin{aligned}
3 * 5 &= 3 + (3 * 4) \\
&= 3 + (3 + (3 * 3)) \\
&= 3 + (3 + (3 + (3 * 2))) \\
&= 3 + (3 + (3 + (3 + (3 * 1)))) \\
&= 3 + (3 + (3 + (3 + (3 + (3 * 0))))) \\
&= 3 + (3 + (3 + (3 + (3 + (0)))))
\end{aligned}
$$

where each line is a new level of recursion.

It would also be good to look at non-arithmetic operators as examples. Below is a basic module for lists:

```
fmod LIST is
   sorts Elt List .
   subsort Elt < List .
   op nil : -> List [ctor] .
   op __ : List List -> List [ctor assoc id: nil] .
endfm
```

Now say we wanted to add an operator size that would return a `Nat` value equal to the number of elements in a list.

```
fmod LIST-SIZE is
   protecting LIST .
   protecting PEANO-NAT .

   op size : List -> Nat .
   var E : Elt .   var L : List .

   eq size(nil) = 0 .
   eq size(E L) = s(size(L)) .
endfm
```

This is a common recursive strategy in modules involving lists: represent the list as its first element concatenated with the rest of the list, perform a single operation on the first element, then call itself recursively on the rest of the list. In this case, we split the list into its first element `E` and the rest of the list `L`, then add one `s( )` for `E`, then call size on `L`.

Recursion is not necessary in every single operation: recall the `author` operation in our card catalog example. There are equations like this that only access a piece of data in a containing structure. In this case, the containing structure is the `n_t_pd_l_` constructor, and the piece of data is the first argument. If one were to represent a deck of playing cards, one might employ the definitions below:

```
fmod CARD-DECK is
   sorts Number Suit Card .
   ops A 2 3 4 5 6 7 8 9 10 J Q K : -> Number [ctor] .
   ops Clubs Diamonds Hearts Spades : -> Suit [ctor] .
   op _of_ : Number Suit -> Card [ctor] .
   op CardNum : Card -> Number .
   op CardSuit : Card -> Suit .
   var N : Number .      var S : Suit .

   eq CardNum( N of S ) = N .
   eq CardSuit( N of S ) = S .
endfm
```

Note that with the examples in lists and cards, it is hardly ever useful to represent a higher-order sort, such as a list or a card, as one variable, but rather as components combined by a constructor. This makes recursive strategies much easier also. Also note that this really only works with constructors. It would be superlatively stupid to represent a number M as a sum made by the addition operator, but not so to represent a number as the successor of M.

Some of the equations are similar to common mathematical properties, in addition to being good simplification rules. For example, the first equation in the Peano notation example, `N + 0 = N`, is the additional property of zero. One will notice that other elementary, common mathematical properties, those of associativity, commutativity, and identity are not expressed as equations; rather, these have special flags. Imagine how one might declare an equation for the commutative property: `N + M + O = N + O + M`. Now, is the right side at all simpler than the left side? Or are they equally simple? (Please answer 'yes' to the second question.) Maude cannot use this equation to simplify, because one side is not simpler. The same goes for commutativity. The end result of these non-simplifying equations is non-termination, which means that the interpreter, trying to reduce the expression, would continually try to reduce to an equally simple expression. Non-simplifying equations cause this, and so Maude designates special flags for the compiler to deal with this problem in an ingenious and, to this tutorial, irrelevant, way.

Identity properties are often useful to define by equations, sometimes even with constructors. `N + 0 = N` and `L nil = L` are both identity attribute equations. We can do this with non-constructors (*defined* operations) because they rarely cause non-termination, so the first equation is valid, but as we will see soon, the second equation may cause non-termination in some cases and should be avoided in favor of the `id:` flag.

Just because things aren't complicated enough yet, I'll add in idempotency. Remember idempotency? It was touched upon in the first chapter. Idempotency is a property belonging to sets, which are multisets (remember that multisets are lists where order doesn't matter, that is, the commutativity property reigns) that don't have more than one of the same elements. For example, a multiset could consist of { 1 , 2 , 2 , 2 , 7 , 7 } but an analogous set would only include { 1 , 2 , 7 }. To express this property using the common list/multiset/set notation of the `__` operator, one has an equation

```
var E : Elt .
eq E E = E .
```

Note that in this case, the right hand side is definitely simpler than the left hand side. Recall that the `idem` flag cannot be used alongside the `assoc` flag, so if you want both properties, you'll need the above equation to declare idempotency. It's also good to note that both identity and idempotency (whether you use the attribute flags or not) can easily become pitfalls of non-termination in operations that use recursion: a list-size operation `length` that is defined as `length(L1 L2) = length(L1) + length(L2)` will run into trouble when `L2` is the `nil`:

length(L1 nil) = length(L1) + length(nil)
= length(L1 nil) + length(nil)
= length(L1) + length(nil) + length(nil)
= …

Because this algorithm splits the list into two arbitrary parts, as opposed to a single element and "the rest," it does not terminate. These pitfalls are easy to repair but hard to spot ahead of time. Generally, we want a recursion that goes in one progressive direction, as opposed to the above, which goes in many directions.

2.2 Conditional Equations

Congratulations! You've read through the toughest part of this chapter. The next topic, if you've understood equations, is a piece of cake: conditional equations. Conditional equations are equations that depend on a Boolean statement. They are written with the key word `ceq` and, after the equation, a condition starting with the key word `if`. Maude comes with a sort `Bool` already provided, along with constants `true` and `false`, and ==, =/=, `and`, `or`, and `not` operations (all are mixfix). A conditional equation will execute a reduction only if its condition reduces to `true`.

```
ceq different?( N , M ) = true if N =/= M .
```

If we want to include more than one condition in a conditional equation, we string them along using the `_/\_` condition-concatenation operator:

```
ceq bothzero?( N , M ) = true if N == M /\ M == 0 .
```

These are a trivial examples, of course. A side note: one thing to look out for when dealing with `Bool`s is the lack of provided comparison operations (<= , >=, <, >). Because each sort has unique definitions for these comparisons, these operations must be defined in the module.

```
fmod NAT-BOOL is
   protecting PEANO-NAT .
   op _<=_ : Nat Nat -> Bool .
   op _>_ : Nat Nat -> Bool .
   vars M N : Nat .
   eq 0 <= N = true .
   eq s(M) <= 0 = false .
   eq s(M) <= s(N) = M <= N .
   eq M > N = not( M <= N ) .
endfm
```

No, this example does not make use of conditional equations. But one usually needs a module like this to use conditional equations in meaningful ways. With the above definitions, I can write an equation for a subtraction operator within the natural numbers (remember that natural numbers are all non-negative).

```
ceq N - M = 0 if M > N .
```

This is an important equation to have if the subtraction operator is to stay within the bounds of the natural numbers.

One more side note: conditional equations are not to be confused with the operator `if_then_else_fi`, an operator provided by Maude (like `Bool`, no importation necessary), which takes a boolean condition as its first argument, then two expressions as the next arguments. It is not necessary to declare equations with this operator as conditional, because they aren't using the *key word* `if`.

```
max( M , N ) = if N > M then N else M fi .
```

One of the most powerful features of Maude 2.0 is the ability to use a pattern match as a condition, using the key symbol `:=`. This symbol compares a pattern on the left-hand side with the right-hand side, and if they match, returns `true`. For example, let's consider an irritable professor who takes questions only after class and simply ignores any other sort of interruption. We'll use the `STRING` module and the operators `_?` and `_!` to denote the difference between a question and an exclamation.

```
fmod IRRITABLE-PROFESSOR is
   protecting STRING .

   sorts Question Exclamation Interruption .
   subsorts Question Exclamation < Interruption .

   op _? : String -> Question [ctor] .
   op _! : String -> Exclamation [ctor] .
   op reply : Interruption -> String .

   var I : Interruption .
   ceq reply(I) = "Questions after class, please"
                                    if (S:String) ? := I .
endfm
```

There are a few things to point out, besides the always fun use of on-the-fly variables: first, we put the pattern in the left-hand side, and the target in the right hand side. Note also that the above could be written as a normal equation

```
eq reply( (S:String) ? ) = "Questions after class, please" .
```

with the same results. The difference is in the ideology: in the pattern-matching example, we're essentially assigning a certain pattern to `I`, then searching for all instances of this pattern in an expression, replacing them with '`I`', and reducing it according to the equation whose left-hand side is `reply(I)`. We can also use the `:=` as a sort of value assignment operator:

```
var T : String .
```

```
ceq reply( (S:String) !) = T if T := "Please be quiet!" .
```

in which case, `T` is the pattern, and wherever it is matched, `"Please be quiet!"` takes its place. We're basically saying "let `T` be equal to `"Please be quiet!"` and go from there." It's essentially, all the same sort of matching, but with a wildly different effect, and it is important not to confuse these two uses. As we can see, the freedom to match patterns in conditions can lead to some exciting and powerful implementations.

2.3 The Maude Environment

Now, the Maude environment, for our present purposes, revolves around the provided command `reduce` (or `red`). First, one must write the modules for the algebras to be used in the environment, either by typing them directly into the prompt (not recommended), or by storing them in a file directory that the Maude environment can access, and then typing `load MODULE-NAME` into the prompt. Then, one types in `reduce`, followed by the expression to be reduced. For example:

```
Maude>  reduce s(0) + s(s(0)) .
result  Nat: s(s(s(0)))
```

Make sure the expression is followed by a period, and that the current module contains the equations necessary to reduce the expression. To change the current module, type in `select MODULE-NAME` into the prompt. Alternatively, one may specify which module contains the rules to be applied to the expression by including the optional key phrase `in MODULE-NAME :`. To whit,

```
Maude>  red in PEANO-NAT-MULT : s(s(0)) * s(s(s(0))) .
result  Nat: s(s(s(s(s(s(0)))))) .
```

Any expression which follows the syntax of a module can be reduced using this command, as long as the module is already loaded. Of course, if an expression is already at its simplest form, the result will simply be the original expression.

For those interested in seeing exactly how an expression is reduced, step by step, by the Maude interpreter, type:

```
Maude> set trace on .
```

Now, whenever you reduce an expression, Maude will take you through each reduction, giving you the equations used, how each variable in the equation was filled, etc. This can be especially useful for debugging a tricky module. For reasons that may become clearer in Chapter 4, this author does not recommend using `trace` while in Full Maude yet, unless you can read really really fast.

2.4 Membership Axioms

Now, we've actually looked at some expressions in Maude that deal with membership logic. "Membership" simply refers to how certain terms are "members" of sorts. When we declare a variable, we declare it as a member of a sort using the colon, which one can think of a symbol for "is a member of." Thus, the declaration `var N : Nat` is the same as saying "variable `N` is a member of the sort `Nat`." But in fact, membership logic is at the bottom of pretty much every declaration in Maude. For example, the declaration of an operation

```
op _+_ : Nat Nat -> Nat .
```

can be expressed as a *conditional membership axiom*. Conditional membership axioms are just like conditional equations, except that they deal with membership, not simplification, and use different key symbols like the colon and key words like `cmb` and `mb` (for conditional membership axiom and membership axioms, respectively). The same idea at work in the above operation declaration can be expressed as

```
cmb N + M : Nat if N : Nat and M : Nat .
```

I don't mean that the above conditional membership axiom replaces the operation declaration, but rather that the two express the same things. Likewise, the declaration of `NzNat` (non-zero natural numbers) as a subsort of `Nat` can be written two different ways.

```
subsort NzNat < Nat .
```

or

```
cmb N : Nat if N : NzNat .
```

The above examples illustrates the idea commonly called "syntactic sugar:" of special key words as programming shortcuts, expressing in simpler terms what can be expressed in a more basic logic. And since we do have this syntactic sugar, in my opinion, we should use it as much as possible. So when are membership axioms and conditional membership axioms useful? An excellent example can be seen if we declare a division operator. To avoid division by zero, we have the second argument of the operator be a non-zero `Nat`, a `NzNat`.

```
sorts Zero NzNat Nat .
subsort Zero NzNat < Nat .
op 0 : -> Zero [ctor] .
op _/_ : Nat NzNat -> Nat .
```

So far, so good, right? But what happens if we enter in the expression `s(s(0)) / (s(s(s(0))) / s(0))` into the Maude environment? This expression, ignoring Peano notation, is 3 / (3 / 1), and should come out as 1. But (3 / 1) returns, according to the declaration, a `Nat`, and therefore it cannot serve as the second argument of another division operation; it must be a `NzNat`. "3 / (3 / 1)" will come out as an error term of kind `[Nat]`. So what we can do is create a conditional membership axiom

```
var N : Nat .       var M : NzNat .
cmb N / M : NzNat if ( N =/= 0 ) .
```

or, alternatively, an equivalent membership axiom:

```
vars N M : NzNat .
mb N / M : NzNat .
```

Either way, (3 / 1) will now be parsed as a `NzNat`, solving our argument problem.

Other common uses of membership axioms appear when importing and building on other modules. For example, if we were to define a module for the card game "crazy eights," where any card with value eight is a wild card, we might write:

```
fmod CRAZY-EIGHTS is
   protecting CARD-DECK .
   sort WildCard .
   subsort WildCard < Card .
   var C : Card .
   cmb C : WildCard if CardNum(C) == 8 .
endfm
```

Or, if for some reason we wanted to define a certain sort `SuicideKing` (if you look at the King of Hearts in pretty much any card deck, he seems to be sticking his sword into his head, hence the name), we might write:

```
fmod SUICIDE-KING is
   protecting CARD-DECK .
   sort SuicideKing .
   subsort SuicideKing < Card .
   mb K of Hearts : SuicideKing .
endfm
```

The matching condition can also be used in conditional membership axioms: consider the following module for poker pairs.

```
fmod CARD-PAIR is
   protecting CARD-DECK .
   sorts Pair PokerPair .
   subsort PokerPair < Pair .
   op <_;_> : Card Card -> Pair [ctor comm] .
   var N : Number .  var P : Pair .
   cmb P : PokerPair if < N of S:Suit ; N of S':Suit > := P .
endfm
```

In this example, the `<_;_>` makes a general pair of any two cards. If the number values of these cards are the same, then they are a pair in the poker definition of the term, or, as above, a `PokerPair`.

2.5 Operator and Statement Attributes

Aside from what we already know of constructors and flags, the user may employ several powerful techniques when declaring operators, which decide how the operator will behave within certain expressions, equations, and reductions. Furthermore, equations themselves may carry flags, or *attributes*, within brackets following the declaration. An examination of operator and statement attributes will prove practicably useful only in specific situations, but lend a more complete picture of the capabilities of Maude.

One extremely powerful operator attribute is the `memo` flag. When Maude comes across an operation with `memo` among its attributes, it "memorizes" the reduced form of any expression with that operator at the top (that is, if the expression were written in prefix mode, the outermost operator). Whenever such an expression appears, Maude refers to its memorization table and produces the reduced form much quicker than if it had to continually apply reduction equations over and over again. This is useful when we write programs where the same expression (important: the same *expression*, not just the same *operator*) pops up thousands and thousands of times, such as highly recursive number theory problems.

When there is any fear of ambiguous expressions – for example, consider the expression `3 + 3 * 3`, which can be understood by Maude as either `(3 + 3) * 3`, that is, `24`, or `3 + (3 * 3)`, that is, `12` – then the user may declare *precedence* for the operator using the flag `prec` and a natural number. Precedence is kind of like the operator's place in line: the lower the number, the higher the precedence, the sooner the operation will be executed. If we declare

```
op _+_ : Num Num -> Num [prec 35] .
op _*_ : Num Num -> Num [prec 25] .
```

then `3 + 3 * 3` will always be `12`. If the user chooses not to give precedence for an operation, Maude secretly and automatically assigns it a precedence value of 41. Precedence can always be overridden with parentheses.

Of course, the expression `3 + 3 + 3` would still be ambiguous. True, we could declare the `_+_` operator with the `assoc` tag, but what about non-associative operators? What if we have, say, a "versus" operation, `_vs_`, which pits two tennis players against each other and returns the winner? How does Maude parse `sampras vs agassi vs roddick`? Suppose we declare the following:

```
op _vs_ : Player Player -> Player [comm] .
ops sampras agassi roddick -> Player .

eq roddick vs agassi = roddick .
eq agassi vs sampras = agassi .
eq sampras vs roddick = sampras .
```

Then `sampras vs (agassi vs roddick)` will result in Pete Sampras as the victor, while `(sampras vs agassi) vs roddick` will give Andy Roddick the cup. In this case, we declare a *gathering pattern* using the flag `gather` and the key symbols `e`, `E`, and `&`.

Still with us? Good for you. The flag we declare looks like `gather (e E &)`, where the number of symbols in the parentheses is the number of arguments the operator takes. If the symbol corresponding to an argument is `e`, this means that the top operator involved in the argument expression must have a precedence value strictly less than that of the operator; if the symbol is `E`, then the argument's precedence may be less than or equal to that of the operator; if the symbol is `&`, Maude accepts an argument of any precedence.

This probably makes no sense yet. Well, for one, it only makes sense for expressions that string operators together. `X + Y` makes no use of the `gather` flag because its arguments are just variables. `X + Y + Z`, however, involves the addition of a variable to another addition: it is either `_+_(X, _+_(Y, Z))` or `_+_(_+_(X, Y), Z)`, depending on the `gather` flag. If, say, we were to declare the `_+_` operator with the flag `gather (E e)`, then only the latter is possible. Why? Because the second underscore of `_+_` can only be filled with an operator of a *lower* `prec` number, which excludes itself. So, the nested operator goes in as the *first* argument, and the second argument takes the variable `Z`. As one can imagine, the default of any declared operation is the `&` in every argument.

Returning to tennis, let us declare the versus operation as

```
op _vs_ : Player Player -> Player [prec 33 gather (e E)] .
```

so that the matches are played starting with the last two players and proceed to the first. Only an event of higher precedence (that is, lower `prec` value) disrupts this order. If we, say, declare the two operators `murders` and `teases` with `prec` values 32 and 34, respectively, then `sampras murders agassi vs roddick` becomes `sampras vs roddick`, while `sampras teases agassi vs roddick` still pits Andre Agassi against Andy Roddick.

Only one statement attribute stands out among the others in its helpfulness (the others, for the most part, become useful only at the meta-level; see Chapter 6), and this is the `owise` flag. Whenever we declare an equation or membership axiom, we set up the left-hand side as a pattern that must be matched in order to reduce to the right-hand side. If we write a second equation, directly following the first, and add the `otherwise` or `owise` flag in brackets, this new statement acts like a conditional equation whose condition is the exact opposite of the first equation. Returning to our irritable professor, if we wanted to make him even brusquer, we could write:

```
eq reply(I) = if I :: Question
              then "Questions after class, please"
              else "Shut up!" fi .
```

Or we could use a conditional equation and `owise`:

```
ceq reply(I) = "Questions after class, please"
                                    if (S:String) ? := I .
eq reply(I) = "Shut up!" [owise] .
```

Yes, the difference is trivial for easily-expressed conditions, but it's rather useful in cases where one side of the condition is much easier to express than the other:

```
eq suicideking?( K of Hearts ) = true .
eq suicideking?( C:Card ) = false [owise] .
```

There are hosts of other nifty little flags, but they escape the depth of this tutorial. The complete list may be found in the Maude 2.0 Manual, Chapter 4, Section 4.

As said before, the above techniques are useful only in specific cases, and do not affect the basic concepts of operations and equations. On the other hand, these tools are powerful, and can make all the difference in more complicated problems.

2.6 Expanding on Chapter One

I always try to keep my promises: in the last chapter, we said we'd elaborate on importing modules, on the usefulness of kinds, and on operator overloading. These fundamental concepts can only be completely understood with reference to equational reduction, and so we have postponed the full story until now.

In Chapter One, we briefly touched upon the three types of module importation: `protecting`, `extending`, and `including`. The basic idea of the `protecting` importation is the same: we don't want to change the meaning of the imported module. We can change this meaning in two key ways: by adding new ground terms (constructors and constants) to a module's sort(s), and by redefining the already extant terms of the imported module. The first is called "junk" and the second, "confusion." For example, if we were to import the module `PEANO-NAT-EXTRA` and declare a new constant `5`,

```
op 5 : -> Nat .
```

this would be junk. The equations in `PEANO-NAT-EXTRA` have nothing to do with `5`, but there it is, stuffed in with the other ground terms `0` and `s`. On the other hand, if we were to redefine the addition operator with the equation

```
eq 5 + N = s( s^4(0) + N ) .
```

or even just

```
eq s^5(0) + N = s( s^4(0) + N ) .
```

that would be confusion, because we're adding reduction rules that mess with the module's original idea of how to reduce the addition operator. Now, Maude will let you do this, but it's

just not *nice*. It's a violation of the imported module's original sense. There will be times when we'll want to change this sense, to be sure, but for the other times, we forbid the Maude compiler to accept junk and confusion by importing the module with the key word `protecting`.

The `extending` importation allows junk, but no confusion. Say we wanted to use `PEANO-NAT-EXTRA` in a module that deals with multivariable algebra. To allow for an expression like X + Y + 8, we'd have to declare the following:

```
sort Variable .
subsort Variable < Nat .
ops X Y : -> Variable [ctor] .
```

This adds junk to the sort `Nat`, because `X` and `Y` are new ground terms that the equations of `PEANO-NAT-EXTRA` fail to deal with (note that, if `Nat` were declared a subsort of `Variable`, this would not be the case, and we could use `protecting`... but that's another story). Anyway, we'd import the module using `extending`, since we don't add any confusion, and we don't want Maude to let us if we make a mistake.

The `including` importation allows junk and confusion. Let's say we import the module `CARD-DECK` into a new `POKER-DECK` module, and add the following declarations:

```
op anyNumber : -> Number .      op anySuit : -> Suit .
op joker : -> Card .
eq CardNum(joker) = anyNumber .
eq CardSuit(joker) = anySuit .
```

This is a perfectly reasonable thing to do, but it adds junk (`anyNumber`, `anySuit`, and `joker`) and confusion (the two new equations redefine `CardNumber` and `CardSuit`). To allow this, we say

```
including CARD-DECK .
```

and we've given fair warning to Maude that we'll be changing the sense of the module.

So much for importations. On to kinds.

In the last chapter, we used a `breed` operation to illustrate the relationship between error terms and kinds, showing that a mismatch of sorts and operators within a connected component returns an error term included in the corresponding kind. We also mentioned that, in certain cases, we may be able to reduce error terms into non-error terms. Intuitively, this makes sense: if we were to create an operator `sire` that returns the first argument of the `breed` function, then `sire(breed(collie, penguin))` should reduce to `Dog : collie`, even though the argument of `sire` is an error term. Consider the following dog-racing module:

```
fmod DOG-RACING is
   protecting DOGS. protecting NAT .
   op race : Dog Dog -> Dog .
```

```
    op speed : Dog -> Nat .
    vars N M : Dog .
    eq speed(bloodhound) = 20 .
    eq speed(collie) = 25 .
    ...
    eq speed(frog) = 5 .
    eq speed(breed(N, M)) = (speed(N) + speed(M)) quo 2 .
    ceq race(N, M) = N if speed(N) > speed(M) .
    eq race(N, M) = M [owise] .
endfm
```

where I have omitted the other dogs' speeds so to save space. The algorithm is simple: the faster dog wins, speed being determined by the average of the speeds of the parents. As the module stands now, the expression `race(breed(collie, frog), bloodhound)` would not reduce, but instead return itself as `[Animal] : race(breed(collie, frog), bloodhound)`. If we bring the equations to the kind level, however, simply by making `N` and `M` variables of the kind `[Dog]` instead of the sort `Dog`, the expression does indeed reduce (to `Dog : bloodhound`). Should we race a collie-frog versus a bloodhound-frog, Maude will declare the `[Animal] : breed(collie, frog)` the winner, *even though it is an error term*. The presence of kinds in Maude allows us to play around with error expressions in ways that will often prove constructive.

Maude allows *overloaded* operators; that is, we can define two different operators with the same name. For example, we can define two _+_ operators, for natural numbers and integers, or for musical notes and all sorts of wild things.

```
op _+_ : Integer Integer -> Integer .
op _+_ : Nat Nat -> Nat .
op _+_ : Note Note -> Chord .
op _+_ : Wrong Wrong -> Right .
```

Note that, since `Nat` is just a subsort of `Integer` (er... you can assume that I've declared this earlier), the second line is just a specification of the first, telling Maude that if we add two natural numbers, the result will not be just an `Integer`, but a `Nat`. This is called *subsort overloading*; the other examples are called *ad-hoc overloading*. In general, it's a bad idea to try ad-hoc overloading constants (guess why), but it can be done by enclosing the constant in parentheses and attaching a dot with the sort name to it within the expression. So, to differentiate between the `Letter A` and the `Note A`, we write `(A).Letter` and `(A).Note`. Subsort overloading requires that both operators be declared with the same equational attributes (the same flags), save for the `ctor` flag.

In the chapter on foundation, the hardest part of learning a term or idea was memorizing the syntax and maybe a bit of the ideology behind it. For equations and membership axioms, the hardest part is the strategy involved in writing them, so that they define tight, complete algebras. I've tried to include some common strategies and examples of how these guidelines can be applied, but only experience, both personal and observational, will teach effectively.

EXERCISES FOR CHAPTER 2

1. There are four, actually five basic requirements for an equation. It must begin with the keyword _____, it must have the key symbol _____ somewhere in it, it must have a _____ at the end, and it must have within it mathematical phrases consisting of variables and _____. Finally, for an equation to be worth its weight, the right-hand side of the equation must be _____ than the left-hand side.

2. Decide what the following expressions are by writing in the necessary keyword:

   a) _____ `N mod M = (N - M) mod M if M <= N .`

   b) _____ `N mod M : NzNat if N =/= M`
   `/\ (N - M) mod M :: NzNat .`

   c) _____ `N mod M = (N - M) mod M .`

   d) _____ `s(0) mod M : NzNat .`

   e) _____ `N mod M = if M <= N then (N - M) mod M else N fi .`

3. The modulus operator `mod` returns the remainder of an integer division: for example, 11 `mod` 3 = 2, 4 `mod` 3 = 1, and 30 `mod` 8 = 6. Given the following module (for the `NAT-BOOL` and `PEANO-NAT` modules, see 2.2 and 1.4, respectively),

```
1       fmod MODULUS is
2            protecting NAT-BOOL .
3            sorts NzNat Zero .
4            subsorts NzNat Zero < Nat .
5            op 0 : -> Zero [ctor] .
6            op _-_ : Nat Nat -> Nat [right id: 0].
7            op _mod_ : Nat NzNat -> Nat [.....] .
8            vars X Y : Nat .    var M : NzNat .
9            cmb X : NzNat if X =/= 0 .
10           eq s X - s Y = if s X > s Y
11                          then X - Y else 0 fi .
```

```
12          ..........................................
13          endfm
```

a) Which expression from question #2 would best fill in the dotted blank on line (12) if you want to reduce the following statement?

```
red in MODULUS
   ( s s s s s 0 mod s s s 0 ) mod s s 0 .
```

_____

b) Write the equation that the identity flag on line (6) replaces.

_____

c) Lines (10) and (11), which, along with line (6), define the subtraction operator, could be replaced by the following two equations:

```
ceq s X - s Y = X - Y if s X > s Y .
eq ..... = 0 [.....] .
```

Choose the expression-and-flag pair that best fills in the two dotted blanks.

a.	`s X - s Y ... memo`	c.	`s X - s Y ... owise`
b.	`X - Y ... owise`	d.	`X - Y ... gather (& e)`

d) The constant operator `0` in line (5) is an example of _____ overloading.

e) What flag(s) should fill the dotted blank on line (7) so that 14 `mod` 5 `mod` 3 reduces to `0`?

a.	`assoc`	c.	`gather (E e)`
b.	`gather (e E)`	d.	`gather(& &)`

4. The sort `Deck` is a list of cards (refer to the `CARD-DECK` module in 2.1) defined by the subsort declaration

```
subsort Card < Deck .
```

and the operator __, defined by

```
op __ : Deck Deck -> Deck [ctor assoc id: null] .
```

```
op null : -> Card [ctor] .
```

a) Write an equation that defines the operation `topcard`, which takes a `Deck` and returns the top card (i.e., the first element.) Assume that the variable `C` of sort `Card` has already been declared.

b) Let sort `FemCard` be defined with the subsort relation `FemCard < Card`. Write one (or more) membership axiom(s), conditional if you wish, that makes a `Card` a `FemCard` if the suit is `Hearts` *or* the card is a Queen. You might want to use the operations `CardNum` and `CardSuit` defined in `CARD-DECK`, but then again you might not. Feel free to declare your variables any way you like.

c) Write one or more equation(s) that define(s) the operator `FirstFemCard`, which returns the first `FemCard` drawn from a deck. You *must* use recursion.

   i. Write it using the if_then_else_fi operator.

   ii. Write it using a conditional equation and the `_::_` operator.

   iii. Write it using a conditional equation and pattern matching.

3. SYSTEM MODULES

3.1 Rewrite Laws

Equations are extremely useful for creating and mapping out structures – structures like a set of operators on constants and sorts, structures like lists and decks of cards – but the real power of Maude is about transitions that occur within and between structures. These transitions are mapped out in rewrite laws. Rewriting logic, like most things in mathematics, is a pretty simple thing unless one tries to explain it in math language. Rewriting logic consists of two key ideas: *states* and *transitions*. States are situations that, alone, are static, and transitions are the transformations that map one state to another. For example, if it's a sunny day, but a big grey cloud comes along and it starts raining, the two *states* would be "It's a sunny day" and "It's a rainy day" and the *transition* would be the rain cloud. A rewrite law declares the relationship between the states and the transitions between them.

Rewrite laws are usually very precise, in the sense that an equation's power lies in its *breadth* of applicability. We could not, for example, have "sunny day" = "rainy day" as a rewrite law, because 1) by itself, it's just not true: a sunny day is not the same thing as a rainy day, and 2) this includes nothing about the transition. Therefore, in Maude, we find it very helpful to give a rewrite law a name in brackets, though this is optional. We might say [rain cloud] : "sunny day" = "rainy day" then. Note that we do not, actually. I'll get to syntax soon.

But probably the most important characteristic of rewrite laws is their irreversibility. It's a one-way street. This is not to say that one state cannot transition to another state and transition back; after all, a rainy day can certainly turn sunny. The difference is that any particular rewrite law cannot go backwards; a rainy day can't turn sunny again *because of a rain cloud*. To make this distinction, we give rewrite laws a special symbol "=>" to show their unidirectional property.

Now for the syntax: we already know about the brackets with the name and the => symbol. We declare a rewrite law with the key word `rl`.

```
rl [raincloud] : sunnyday => rainyday .
```

For an equation to make sense, both sides of the equation have to return the same sort. This comes as intuitive for equations; after all, for something to equal another they should be of the same sorts. Thus, the fact that both sides of `s(M) + N = s(M + N)` are of sort `Nat` comes, er, naturally. Well, the same thing goes for rewrite laws. Even though rewrite laws map transitions of states, these transitions are all within an all-encompassing sort. For example, we could have declared the rewrite law in a system module such as:

```
mod CLIMATE is
   sort weathercondition .
   op sunnyday : -> weathercondition .
   op rainyday : -> weathercondition .
   rl [raincloud] : sunnyday => rainyday .
endm
```

Many times, we just call the all-encompassing sort "`State`" and make more specific subsorts.

One thing that you might notice is that we didn't use variables in this rewrite law. Rewrite laws often map transitions between very specific states, which are therefore expressed as constructors, since the states cannot, and should not, be reduced further. Note that a constructor, if it is not a constant, can take an argument, such as an `op talking : Adverb -> State [ctor] .` An example of this state might be "`talking(loudly)`." Variables appear most often when the rewrite law describes a precise transition that may be applied to a certain phylum of states. We'll see some examples of this soon enough.

So now that I've dumped all this information on you, let's look at a concrete example. There's this mind puzzle that I heard when I was a little kid: a certain hobo can make one cigarette out of four cigarette butts (the butt of course, is what's left after smoking a cigarette); if the hobo starts off with sixteen cigarettes, how many cigarettes can he smoke in total? The answer is twenty one, because once he smokes 16 cigarettes, he can make the sixteen butts into 4 more cigarettes, and once he smokes those, he can make the four butts into 1 more cigarette. So how do we implement this puzzle in rewriting laws? Basically, there are two states involved: cigarette and butt (smoked cigarette), and two transformations: transforming a cigarette into a butt (smoking) and transforming four butts into a cigarette (making a new cigarette).

```
mod CIGARETTES is
   sort State .
   op c : -> State [ctor] .    *** cigarette
   op b : -> State [ctor] .    *** butt
   op __ : State State -> State [ctor assoc comm] .
   rl [smoke] : c => b .
   rl [makenew]   : b b b b => c .
endm
```

To actually do something with this module, we use the rewrite command in the Maude environment. There are certain strategies in applying rewrite rules; it may have occurred to you that a module with a lot of rewrite rules might create some confusion between all these states and transitions with little hierarchy between them. Strategies provide an "order of operation" for applying rewrite rules. Creating strategies is much too complex to discuss here, but I will say that the rewrite command is a strategy, a default strategy which Maude provides. One invokes it with the key word `rewrite` (or `rew`) followed by a natural number in brackets representing the maximum number of rewrite laws that you want to be applied. Of course, if you choose to leave out the number in brackets, this becomes an *unbounded* rewrite, and Maude will apply rewrite laws until they terminate. Now, one can imagine a set of rewrite laws such as `rl a => b` and `rl b => a`, a pair of laws that doesn't terminate, but is still a legitimate set of rewrite laws for all that Maude cares. Therefore, we usually want to use the cutoff the rewrite command provides as a sort of "plan B" in case our module's rewrite laws don't terminate by themselves.

So let's plug in "`rew [100] c c c c c c c c c c c c c c c c`" into the Maude environment: this represents 16 cigarettes. The sixteen cigarettes become sixteen butts because

of the `smoking` rewrite law, then four cigarettes because of `makenew`, then four butts because of `smoking`, then one cigarette because of `makenew`, and finally one butt because of `smoking`. So we'll end up with just a "`b`" displayed. Of course, this doesn't really illustrate the solution of the puzzle; it doesn't count how many cigarettes have been smoked in total. We can adjust the module pretty easily:

```
mod COUNTING-CIGARETTES is
   protecting NAT .
   sort State .
   op c : Nat -> State [ctor] .
   op b : Nat -> State [ctor] .
   op __ : State State -> State [ctor assoc comm] .
   vars W X Y Z : Nat .
   rl [smoke] : c(X) => b(X + 1) .
   rl [makenew]     : b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .
endm
```

So if we enter "`rew c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0)`" into the Maude environment, we will end up with "`b(21)`" as our answer.

3.2 The Toy Crane Example

The cigarettes module is a good example with which to break into rewriting logic, but doesn't necessarily provide the best illustration of the *powers* of rewriting logic. For one, in the cigarette example, we see rewriting laws drawing transitions from complex states to simpler states, *and this does not have to be the case*. While in equations our goal was simplify, simplify, simplify, rewriting laws most of the time deal with states neither simpler nor more complex than one another.

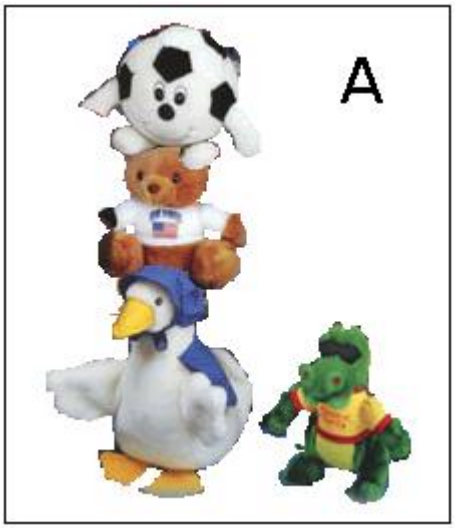
A common and oft-used example in rewriting logic is the blocks world. The basic idea is to create an algorithm for a bunch of wooden blocks stacked on each other or the table, and a robot arm that can carry them. Since it's such a great example, it's also used to illustrate principles of object-oriented programming and artificial intelligence. Here, I'm going to tweak the example to a more familiar situation: those arcade crane machines with stuffed animals in a big glass box and a controllable arm that moves and tries to grab them. I think everyone has been ripped off by one of these at one point in their life, but just in case I'll

provide a picture.

So what states are there? For stuffed animals, there are three state constructors needed: 1) the stuffed animal is on the floor of the machine, not on top of any other stuffed animal; 2) the stuffed animal is on top of another stuffed animal; 3) the stuffed animal is clear, that is, there are no other stuffed animals on top of it. For the robot arm/claw/crane, there are two: 1) the claw is holding a stuffed animal, or 2) it is empty. Any pile of stuffed animals in an arcade crane may be represented as a combination of these states. And we have four main transitions: the claw picks up, puts down, unstacks, or stacks. These last two may seem the same as the first two, but the difference is that a picked up or put down stuffed animal must be on the table, while stacked or unstacked stuffed animals must be on top of another stuffed animal. If this is confusing, perhaps the Maude implementation will make things a bit clearer. And to counteract this possible clarity, we're going to add some `Qid`s in there.

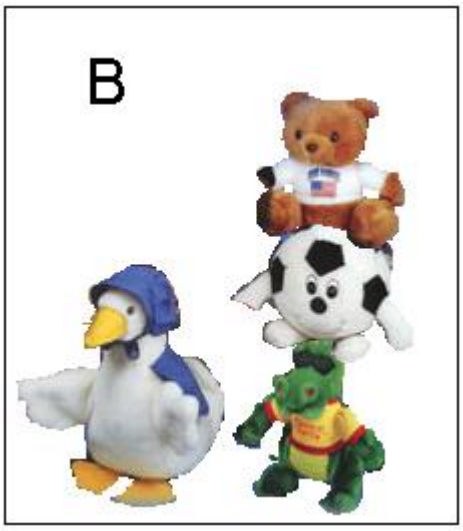
```
   mod ARCADE-CRANE is
      protecting QID .
      sorts ToyID State .
      subsort Qid < ToyID .

      op floor : ToyID -> State [ctor] .
      op on : ToyID ToyID -> State [ctor] .
      op clear : ToyID -> State [ctor] .
      op hold : ToyID -> State [ctor] .
      op empty : -> State [ctor] .
      op 1 : -> State [ctor] .
*** this is the identity State; it's just good to have one.
      op _&_ : State State -> State [ctor assoc comm id: 1] .
      vars X Y : ToyID .

      rl [pickup] : empty & clear(X) & floor(X) => hold(X) .
      rl [putdown] : hold(X) => empty & clear(X) & floor(X) .
      rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
      rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
   endm
```

To look at an example, take a look at the figures below. In box A, the entire state can be expressed as "`empty & floor('mothergoose) & on('teddybear,'mothergoose) & on('soccerball,'teddybear) & clear('soccerball) & floor('dragondude) & clear('dragondude).`"



In box B, the state is described as "`empty & floor('mothergoose) & clear('mothergoose) & floor('dragondude) & on('soccerball, 'dragondude) & on('teddybear, 'soccerball) &`

`clear('teddybear)`." And the set of transitions between these two states would be rewrite law `unstack` called on `'soccerball`, `stack` called on `'soccerball` and `'dragondude`, `unstack` called on `'teddybear`, and `stack` called on `'teddybear` and `'soccerball`. And if we wanted to put the Mother Goose on top of them all, we would have to call `pickup` on her and `stack` on her and `'teddybear`.



Notice that neither State A nor State B is any simpler or more complex than the other. This means that any execution of these laws is potentially non-terminating, that is, there is no specific direction which rewriting will take the states, and might randomly transition ad infinitum. This is when it's important to use the built in limit on the `rew` command.

Also note that in our module, the transitions represented by the rewrite laws are between very precise states. The more realistic the situation, the more factors will be involved in each state, and the more information will be involved in the rewrite laws. For example, we could build upon this example and account for the horribly weak grip of the claw, the fact that some animals would be easier to pick up than others, etc. This would create even more states and therefore even more precise transitions.

3.3 Rewriting in the Maude Environment

We have already seen the `rewrite` operation at work, but the Maude environment provides several other useful tricks to more fully explore the powers of rewrite logic. Now, if we were to enter our `ARCADE-CRANE` module into the Maude environment and call `rew [2]` on the state represented by Box A, we'd find that the default strategy has stacked Dragon Dude on top of the soccer ball. So far, so good. If we call `rew [4]` on the same state, we end up with... Dragon Dude stacked on the soccer ball. In fact, upon further investigation, we'll find that, after the initial `pickup` of Dragon Dude, all that's happening is a continual `stack`ing and `unstack`ing of Dragon Dude! Any even number of `rewrite`s, even as high as thirty or a hundred, will result in the same state as two `rewrite`s.

Obviously, the default strategy has very little imagination. It's really only applying two out of the four rewriting laws. Because such inane loops may often appear in rewriting systems, Maude provides another rewrite command, `frewrite`, or "fair rewrite," which picks which rewrite laws to apply such that no law goes ignored. As it turns out, if we plug in `frew [30]` with our Box A state, this fair strategy ends up scattering the whole stack across the floor. This is certainly much more interesting than the `stack`-`unstack` loop, though perhaps not as interesting as it could be. But hey, `frewrite` is still a default strategy, meant to be applied to any and every rewriting system it comes across. So don't expect it to build the Tower of Hanoi any time soon.

Let's say we wanted to know how quickly `frewrite` scatters the stack on to the floor. We'd start off with

```
Maude> frew [1] empty & floor('mothergoose) &
on('teddybear,'mothergoose) & on('soccerball,'teddybear) &
clear('soccerball) & floor('dragondude) & clear('dragondude) .
...
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result State: floor('mothergoose) & clear('soccerball) &
hold('dragondude) & on('teddybear, 'mothergoose) & on('soccerball,
'teddybear)
```

The Maude environment always repeats to the user the command it receives, which I have omitted with ellipses. Anyway, the first rewrite shows us that the crane has picked up Dragon Dude. Instead of following with `frew [2]` of the same state, we use the provided `continue` command, which continues the current rewrite strategy for X more rewrites, X being a bound supplied by the user.

```
Maude> continue 1 .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result (sort not calculated): floor('mothergoose) &
clear('soccerball) & (empty & floor('dragondude) &
clear('dragondude)) & on('teddybear, 'mothergoose) & on('soccerball,
'teddybear)
```

You don't need to worry about that `sort not calculated` line; as one can see from the extra parentheses, Maude hasn't exactly finished processing the result, but it won't interfere with our rewriting. In any case, we see that the crane just set down Dragon Dude. If we keep entering in `continue 1`, we find that it takes eight rewrites to get all the stuffed animals on the floor. Then, for fun, we enter in `continue 100` and see that they're... still all on the floor. Oh well – it's a default strategy, after all.

One extremely powerful tool provided by Maude is the `search` command. Just like it sounds, it searches for paths of rewrite laws between a beginning and end state supplied by the user. For example, I wanted to find out how, beginning with our Box A state, we could switch Teddy Bear and Mother Goose so that instead of `on('teddybear, 'mothergoose)` we have `on('mothergoose, 'teddybear)`. So, I entered into Maude the following and got:

```
Maude> search in ARCADE-CRANE : empty & floor('mothergoose) &
on('teddybear, 'mothergoose) & on('soccerball, 'teddybear) &
clear('soccerball) & floor('dragondude) & clear('dragondude) =>+
empty & floor('teddybear) & on ('mothergoose,'teddybear) &
on('soccerball,'mothergoose) & clear('soccerball) &
floor('dragondude) & clear('dragondude) .
...

Solution 1 (state 69)
states: 70  rewrites: 125 in 0ms cpu (0ms real) (~rewrites/second)
empty substitution
```

```
No more solutions.
states: 125  rewrites: 272 in 10ms cpu (10ms real) (27200 rew/s)
```

First, some syntax: the `search` key word is obviously necessary, but one doesn't need to explicitly name the search module, as long as the currently selected module is the correct one, but if you leave out the module name, drop `in` and the colon as well; the `=>+` symbol means that you are looking for a search solution involving at least one rewrite law. The other commonly used symbol is `=>!`, which means you're looking for a terminal state, that is, something that can't be rewritten any more. But since the crane system is non-terminating, we use the first option.

The empty substitutions means that we haven't involved any variables in our search. If we wanted to search for a path from our Box-A-state to *anything* with the crane empty and Mother Goose and Teddy Bear on the floor and clear, we'd enter:

```
Maude> search in ARCADE-CRANE : empty & floor('mothergoose) &
on('teddybear,'mothergoose) & on('soccerball,'teddybear) &
clear('soccerball) & floor('dragondude) & clear('dragondude) =>+
empty & floor('teddybear) & floor('mothergoose) & M:State .
```

Don't forget to declare the variable `M` on-the-fly, or the command won't parse. The result of this is three different solutions:

```
Solution 1 (state 10)
states: 11  rewrites: 16 in 0ms cpu (0ms real) (~rewrites/second)
M:State --> floor('dragondude) & clear('soccerball) &
      on('soccerball, 'dragondude)

Solution 2 (state 15)
states: 16  rewrites 23 in 0ms cpu (0ms real) (~rewrites/second)
M:State --> floor('soccerball) & floor('dragondude) &
      clear('soccerball) & clear('dragondude)

Solution 3 (state 34)
states: 25  rewrites: 65 in 0ms cpu (0ms real) (~rewrites/second)
M:State --> floor('soccerball) & clear('dragondude) &
      on('dragondude, 'soccerball)
```

These three solutions represent the three possibilities that involve Mother Goose and Teddy Bear clear and on the floor, where the third line tells the user what state it has plugged in for the variable `M:State`.

Thus, depending on the search, you may or may not get more than one solution. The important thing to look for is the number next to `state` in the parentheses. This number will allow you to see the rewrite path, which is after all what we want. In our first `search` example, a few pages back, we see that Solution 1 is connected to a certain state 69. We type into Maude `show path 69 .` and it displays the path it used to reach the end state:

```
Maude> show path 69 .
state 0, State: empty & ...
===[ rl empty & clear(X) & on(X, Y) => clear(Y) & hold(X) [label
     unstack] . ]===>
state 2, State: floor(‘mothergoose) & ...
===[... [label stack] ...
state 4, State: ...
===[... [label unstack] ...
state 6, State: ...
===[...
state 10, State: ...
===[...
state 17, State: ...
...
state 69, State: empty & floor('teddybear) & floor('dragondude) &
     clear('soccerball) & clear('dragondude) & on('mothergoose,
     'teddybear) & on('soccerball, 'mothergoose)
```

I’ve cut out a lot from the transcription, to highlight the important points. One, you can see how Maude lists the rewrite rule within the `===[ ]===>`, as well as its label, when transitioning to the next state. Also, you might notice that the state numbers skip. The omitted states are the dead-end searches, the rewrites that went nowhere. The idea is that Maude performed one hundred twenty five rewrites and ended up with seventy states, some of which form a solution path. In case you’re wondering, the solution to our question was, in my own abridged notation,

```
[unstack] : on('sb, 'tb) => hold('sb)              (0)=>(2)
[stack] : hold('sb) => on('sb, 'dd)                (2)=>(4)
[unstack] : on('tb, 'mg) => hold('tb)              (4)=>(6)
[putdown] : hold('tb) => floor('tb)                (6)=>(10)
[pickup] : floor('mg) => hold('mg)                 (10)=>(17)
[stack] : hold('mg) => on('mg, 'tb)                (17)=>(28)
[unstack] : on('sb, 'dd) => hold('sb)              (28)=>(48)
[stack] : hold('sb) => on('sb, 'mg)                (48)=>(69)
```

Where the numbers in parens are the states through which the law transitions.

3.4 Conditional Rewrite Laws

Conditional rewrite laws should not be too difficult to explain now that we have both rewrite laws and conditional equations under our belt. Conditional rewrite laws use the keyword `crl`, and the rest is a rewrite law with an `if` statement at the end. Conditional rewrite laws are not extremely common, because usually the pattern of states in the left-hand side of a rewrite law supplies all the “ifs” of the transition. However, sometimes they are undeniably useful. If we wanted to mention the weight of stuffed animal, we might write the following rewrite rule:

```
crl [pickup] : empty & clear(X) & floor(X) => hold(X)
                                          if weight(X) < 10 .
```

This, obviously, assumes there is a `weight` operation which returns the heaviness of the stuffed animals in some unit or another. In general, conditional rewrite laws are used when there's a condition that can't be easily expressed as a state. Like conditional equations, the if statement can take the form of a Boolean statement or a pattern match, but in addition to these, the condition of a rewrite rule may also be *another rewrite rule*. The following conditional rewrite rules are all equivalent:

```
crl [equation1] : a(X) => b(X - 1) if X > 0 .
crl [equation2] : a(X) => b(X - 1) if X > 0 == true .
crl [equation3] : a(X) => b(X - 1) if X > 0 = true .
crl [membership1] : a(X) => b(X - 1) if X :: NzNat .
crl [membership2] : a(X) => b(X - 1) if X : NzNat .
crl [pattern] : a(X) => b(X - 1) if s(N:Nat) := X .
```

If we select one of these rewrite rules and get rid of the others (take your pick), we can write:

```
crl [rewrite] : b(X) => c(X * 2) if a(X) => b(Y) .
```

What in the world does this mean? It means that the rewrite law may be executed on `b(X)` only if `a(X)` could transition to a some state of `b`. Think of the => not so much as the rewrite symbol but as the search symbol: the idea is, the condition is fulfilled if it's *possible* to rewrite the left-hand side into the right-hand side. So, in the above example, while `a(3)` can rewrite to `c(4)`:

`a(3)` => `b(2)` => `c(4)`

the state `a(1)` could never reach `c(0)`:

`a(1)` => `b(0)` =/=> `c(0)`

because `a(0)` cannot transition into any `b` state according to any one of the rewrite laws above.

3.5 Rewrite Laws and Equations

At first glance, the difference between using equations and rewrite laws may seem trivial, but they solve very different problems. Equations are much better than rewrite laws at simplification; however, rewrite laws are better at expressing problems with just one level of simplicity. Rewriting laws also can illustrate constitutional changes that equations can't; though an equation may simplify an expression, the expression is still mathematically equal to its predecessor. A functional module would have a hard time creating the cigarette example, because a cigarette butt is definitely not equal to a newly made cigarette.

In general, equational logic creates a framework through which rewriting laws trace transitions. System modules often include equations; when they do, they set up and define all the operations that become states, and then the rewrite laws deal with these states. For example, in the slightly expanded arcade crane example, we would have to define the operation `weight` using an

equation. We also use equations to define numbers (creating our own notation, such as the Peano notation, or using the library module `INT`, which is also defined with equations) that can later be used in rewrite laws.

Equations are the nuts and bolts and gears and girders; rewrite laws create the machine.

EXERCISES FOR CHAPTER 3

1. While _____ describe rules of simplification (writing the same thing a different way), a _____ maps a _____ between two states (a *change* in constitution). And while the right-hand side of a(n) _____ should be simpler than the left-hand side, this is not necessarily true for the other. This latter uses the key symbol => to emphasize its _____, that is, the fact that it only works in one direction.

   Both equations and rewrite laws may be found in a _____ module, initiated with the keyword _____.

2. The following module sets down the rules for a rewrite game, called "Zip-Zap-Zop." Whenever the constants `zip`, `zap`, and `zop` appear in that order, they become a `win`. And `win`s can change into all sorts of combinations of the three. In addition, four `zop`s in a row ends the game. The goal is, then, to apply the correct rewrite rules so that we end with one `win`. Look over the module and answer the questions that follow:

```
1       mod ZIP-ZAP-ZOP is
2            sort GameState .
3            op _>_ : GameState GameState ->
4                     GameState [ctor assoc id: 1] .
5            op 1 : -> GameState [ctor] .
6            ops zip zap zop : -> GameState [ctor] .
7            ops win gameover : -> GameState [ctor] .
8
9            rl [winner!] : zip > zap > zop => win .
10           rl [scatter1] : win => zop > zip .
11           rl [scatter2] : win => zip > zap .
12           rl [scatter3] : win => zop > zop .
13           rl [gameend] : ..... .
14      endm
```

a) What's missing from line (4) that would allow Maude to rewrite `zop > zap > zip` ?

_____

b) Fill in the dotted blank on line (13):

_____

c) Consider the expression `win > zop > win` :

i. Does an unbounded rewrite of this expression always terminate? _____

ii. What are the five rewrites, in order of application, that must occur in order for the above expression to reach a `gameover`? In the first blank, put the label of the rewrite law you apply, and in the second, the state that results.

`win > zop > win`

1. _____ => _____
2. _____ => _____
3. _____ => _____
4. _____ => _____
5. _____ => _____

d) Let's say we want to add a rule that will change `zap` into `zip` as long as this immediately produces a `win` in the expression. Write a conditional rewrite law `change` that transforms `zap > S` into `zip > S`, and the condition is another rewrite law. Assume variables `S` and `P` of `GameState` have already been declared.

3. Two separate searches were performed with the complete `ZIP-ZAP-ZOP` module (that is, including the conditional rewrite law `change`), and the transcription appears below. The ellipses appear in the place of unimportant information, so don't worry about those, but some important parts have been substituted by dotted blanks. Study the searches carefully and answer the questions below.

First search:

```
Maude> search zap > zap > zip > zap > zop ….(a)…. win .
...

No solution.
states: 22  rewrites: 106 ...
```

Second search:

```
Maude> search zap > zap > zip > zap > zop ….(a)…. win .
...

Solution 1 (state 18)
states: 19  rewrites: 103 ...
empty substitution

No more solutions.
states: 22  rewrites: 106 ...
Maude> show ……(b)…… .
state 0, GameState: zap > zap > zip > zap > zop
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 1, GameState: zap > zap > win
==[ rl win => zop > zop [label scatter3] . ]===>
state 3, GameState: zap > zap > zop > zop
==[ crl ... [label change] . ]===>
state 9, GameState: ……(c)……
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 11, GameState: win > zop
==[ rl ……(d)…… [label ……(d)……] . ]===>
state 16, GameState: zip > zap > zop
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 19, GameState: win
```

a) As you can see, the first and the second searches are almost identical, except for the symbol that goes in the two dotted blanks labeled **(a)**.
   i. What symbol must have been used for the first search? _____
   ii. What symbol must have been used for the second search? _____
   iii. If we had written `zop > zop > zop` instead of `win` in the first search, would Maude have found a solution? _____
   iv. If we had written `zop > zop > zop` instead of `win` in the *second* search, would Maude have found a solution? _____

b) Fill in the dotted blank labeled **(b)**: _____
   If you had entered in `show path 11 .` instead, what would the last line of your result be? _____

c) Refer to question 2.d) and the rest of the search transcript. How should you fill the dotted blank labeled **(c)**? _____

d) Based on the evidence of states 11 and 16, what must go in the two dotted blanks labeled **(d)**? _____
   and
   _____

4. FULL MAUDE & OBJECT-ORIENTED MODULES

4.1 Full Maude

What is Full Maude?

Full Maude is an adventure – more difficult, to be sure, and often even frustrating, but ultimately rewarding. Full Maude extends the capabilities of Core Maude, adding powerful new features that can make all the difference in a project, though at the same time, it complicates or even does away with some of the 'safety nets' that keep a greenhorn Maude programmer from crashing the environment. Therefore, Full Maude requires some finesse and expertise to use effectively, and also some patience, for Full Maude is above all a Great Experiment in progress, an important *tour-de-force* of Maude programming and its flexibility. Even if you choose not to use Full Maude very often, it is essential to *learn* it.

To understand what Full Maude actually is, one must start at the beginning, which is the standard library module `LOOP-MODE`. This module provides the programmer with the tools to create a user interface, so that, in addition to the provided commands of the Maude environment, we can fashion commands of our own for a module-defined "sub-environment" within Maude. The magic of `LOOP-MODE` lies in a single, simple constructor:

```
op [_,_,_] : QidList State QidList -> System [ctor special (….)] .
```

The procedure is this: the user enters in some word or words into the Maude environment, enclosed in parentheses. Because of the parentheses, Maude recognizes the phrase as a loop command, and converts each word into a quoted identifier. This becomes the first `QidList` argument. Some sort of rewriting, defined by the program modules, occurs, usually with the aid of the second argument, a sort `State` declared within `LOOP-MODE`. More on this later. At the end of the rewriting, whatever `QidList` ends up as the third argument is printed to the screen, without the quotes.

Let's look at an example. When I was a kid I used to spend ludicrous amounts of time coming up with ASCII smiley faces, such as: **: –)** (standard), **8 – 0** (gawping), **!: -)** (well-groomed), **[: - |** (Frankenstein's monster), etc. The program below takes in a command from the user – which smiley face to print out – and outputs that particular face.

```
mod SMILE-LOOP is
  inc LOOP-MODE .
  op none : -> State .
  op startsmiling : -> System .
  eq startsmiling = [nil, none, nil] .
  var S : State . vars I O : QidList .
  rl [smile] : ['smile I, S, O] => [I, S, O ' ': '- '`)] .
  rl [wink] : ['wink I, S, O] => [I, S, O ' '; '- '`)] .
  rl [gawp] : ['gawp I, S, O] => [I, S, O ' '8 '- '`)] .
  rl [elvis] : ['elvis I, S, O] => [I, S, O ' '? 'B '- '`)] .
```

```
endm
```

To run the smiley "sub-environment," we first load the above module and then initialize the loop to some starting point with the keyword `loop`. The convention is to define this starting point within the module, as we did above with `startsmiling`. But the following two commands are equivalent and equally valid:

```
Maude> loop startsmiling .
Maude> loop [nil, none, nil] .
```

Now, we simply enter in the commands we want, being sure to enclose them in parentheses:

```
Maude> (smile)
 : -)
Maude> (elvis)
 ? B -)
Maude> (smile elvis gawp)
 : -)  ? B -)  8 - 0
```

A few things to point out: one, we need to backquote the parentheses before making it into a `Qid`, because otherwise Maude thinks it's a real paren and we get a parse error. Also look at the quote standing by itself between the `O` variable and the smiley-face: that's the blank-space `Qid`. Finally, notice that we must import `LOOP-MODE` using `including`, not `protecting`, because we mess around with the definitions of `State` and `System`. Speaking of which, you might notice we made no use of the `State` argument. This second argument can often be very useful for more complicated environments that require a bit more rewriting before it can output an answer:

```
mod MOOD-LOOP is
   inc LOOP-MODE .
   sort Mood .
   op <_;_> : Mood Mood -> State [ctor] .
   op init : -> System .
   ops happy shocked playful : -> Mood .
   op # : -> Mood .   ***blank placeholder
   eq init = [nil, < # ; # >, nil] .

   vars I O : QidList .  var S : State .

   rl [in-smile] : ['smile I, S, O] => [I, < happy ; # >, O] .
   rl [in-gape] : ['gape I, S, O] => [I, < shocked ; # >, O] .
   rl [in-wink] : ['wink I, S, O] => [I, < playful ; # >, O] .

   rl [out-smile] : [I, < # ; happy >, O] =>
                                    [I, < # ; # >, O ' ': '- '`)] .
   rl [out-gape] : [I, < # ; shocked >, O] =>
                                    [I, < # ; # >, O ' '8 '- '0] .
   rl [out-wink] : [I, < # ; playful >, O] =>
                                    [I, < # ; # >, O ' '; '- '`)] .
```

```
    rl [wow!] : < happy ; # > => < # ; shocked > .
    rl [hehe] : < happy ; # > => < # ; playful > .
    rl [hey!] : < playful ; # > => < # ; shocked > .
endm
```

Here, the process is slightly more complicated. First, the user inputs the command, and we convert it into a mood using the `in-` laws. Then, the three last rules describe the mood changes, and lastly, the `out-` laws output the corresponding smiley.

As you can imagine, we can create some amazingly complex and powerful rewriting systems that fit into the `LOOP-MODE` constructor, thereby adding the power of user interaction. In fact, we could even write an entire *programming language* using loop mode, by defining our own commands and making them programming language keywords, then setting down rewrite modules to interpret these keywords into Maude terms and expressions. In fact, we could even write this loopy programming language with the *same keywords as Maude*, plus a few extra ones, so that essentially, all we're doing is giving extended capabilities to the Maude language we already know.

And we did. We called it 'Full Maude.'

Maude is a modeling tool so powerful it can model almost anything, including itself. Here's how it works. Full Maude is an extension of Core Maude written in Core Maude using loop-mode. So when we load a Full Maude module or command, or enter it into the environment directly, the `LOOP-MODE` constructor receives it as input, processes it according to the reduction and rewrite rules set down in the Full Maude signature (a series of modules that mimic and extend the syntax of Core Maude), and then output it in a style similar to the output of Core Maude. The only real visible differences are: 1) the extended features, and 2) we have to put everything in parentheses. Modules, commands, everything we want Full Maude to pay attention to – has to be in parentheses. Other than that, we use the exact same syntax for Core Maude, plus a new set of syntax for the extended features.

What are these extended features? Aside from a few helpful functions here and there, the two main tools of Full Maude are *object-oriented modules* and *parameterization*. The rest of this chapter will explore the syntax and uses of object-oriented programming, and Chapter 5 will tackle parameterized systems. So, to begin.

4.2 Introduction to Object Oriented Ideology

The ancient Greeks introduced the Western world to the concept of the mnemonic, which is any device used to remember a set of data better. One such device involved touring some temple or mansion or public building, and assigning pieces of data to specific things or places within the building. For example, an orator might walk through an old statue garden and assign to each statue a certain point or item to be brought up in his speech. Then, when giving the speech for real, he would walk through the garden in his mind, and discourse on each subject as he encountered its symbolic statue during the mental tour. These "temples of memory" were

commonly used by teachers and politicians and any body who had a lot to memorize. Ironically, these "temples of memory" give the orator *more* information to memorize instead of less. But it still makes remembering easier, because the properties of spatial location lend otherwise abstract concepts a solidity, a realness, and bring out certain traits in these concepts such as hierarchy, order, and inclusive/exclusive qualities.

In programming, one can think of an object as a computer's "temple of memory." An object is simply a structure that groups together data of all types. For example, one could make an object `CAT` which includes a variable for the color of its fur, a variable for its weight, and an operation "`purr`." Now, it is perfectly feasible to work with each operation and each variable separately, but an object establishes implicit connections between each piece of data just by existing and including them. An object also gives the group exclusivity; now `purr` is utterly separate from, say, the addition operator. The structure of objects merely explicitly states for the computer what the programmer already knows. This strengthens, condenses, and simplifies code remarkably.

The big vocabulary word to learn this chapter, which you will encounter in *every* discourse on object oriented programming, is "encapsulation." An object encapsulates data into one easily manageable structure. Encapsulation also carries the connotations of exclusivity and inclusiveness, which are essential to objects. In the next section we'll look at how we actually implement all this ideology in Full Maude.

4.3 Objects in Maude

Objects all belong to a class, which is a general category of which objects are specific instances. For example, the object `CAT` could belong to class `Animal`. Or, class `TABLE` might have instances `Table1`, `Table2`, `Table3`, representing certain tables in a restaurant. The relationship between classes and objects is somewhat analogous to the relationship between sorts and variables of those sorts.

Classes are declared in Maude with a class name and however many attributes you want linked to objects of that class. Attributes are declared with an attribute name and the attribute's sort. Multiple attributes are separated by commas (the set of a class's attributes is commutative, so order does not matter). We also declare classes with the keyword `class` and the | symbol, as follows:

```
class TABLE | occupied : Bool , chairs : Nat .
```

`TABLE` has attributes `occupied`, which is a Boolean value (whether or not the table is occupied), and `chairs`, a `Nat` denoting the number of chairs at the table. Objects are a bit more complicated to declare, because one must declare each individual piece (name and attribute(s)) using variables. For example, an object of `TABLE` needs a name, a Boolean value, and a `Nat`.

```
var A : Oid .  var O : Bool .  var N : Nat .
```

"Oid" stands for "object identifier," which is a fancy name for "object name." Object identifiers are just variable specifically for objects. It comes as a built-in sort in omods. So we've declared variables for all the pieces of a TABLE object; if we want to use this object, say, in a rewrite law, we call it with the symbols < >, | , and the colon.

```
< A : TABLE | occupied : O , chairs : N >
```

This syntax can look pretty baffling at first. If it helps, think of the above structure not as a piece of syntax, but as a constructor defined by Full Maude in its signature:

```
op < _:_ | _ > : Oid Cid AttrSet -> Object [ctor] .
```

Just remember that the | symbol always separates the class/object name from its attributes, and that colons separate attribute names and their associated sorts/values. Objects are above all data *structures*, like computational file cabinets, and the look of the object constructor emphasizes the ideology of encapsulation, that several pieces of information ("occupied : O , chairs : N") are gathered under the same name ("A : TABLE") and separated from the rest of the environment ("< >").

Just as sorts can have subsorts, classes can have subclasses. In C++, certain classes are constructed from other classes, inheriting their attributes. In Maude, the concept of "inheritance" is implemented through classes and subclasses. Suppose our restaurant puts some tables outside, and during the winter we want to know if a certain table is next to a propane heater-lamp. We declare a class OutDoorTABLE, with this additional attribute.

```
class OutDoorTABLE | next2heater : Bool .
```

We then declare OutDoorTABLE a subclass of TABLE. OutDoorTABLE will inherit the attributes of TABLE, in addition to this new attribute next2heater. Since subclasses parallel subsorts, the declaration syntax will not come as a surprise:

```
subclass OutDoorTABLE < TABLE .
```

Note that OutDoorTABLE is as much a class as TABLE; the key word subclass just establishes an inheritance hierarchy. An object of OutDoorTABLE, once all variables are declared, might look like:

```
< A : OutDoorTABLE | occupied : O , chairs : N , next2heater : H >
```

4.4 Messages

If classes are something like bigger versions of sorts, and objects are something like bigger versions of variables, then messages are something like the bigger versions of operations. They are declared with the key word msg, and are otherwise declared pretty much like operations: after msg comes the name (in either prefix or mixfix), followed by a colon, then the sorts fed into the message, then the -> symbol. The two special requirements are that, for messages, the -> symbol is always followed by the key sort Msg, and the first argument is always an Oid, assumed

to be the name of the object upon which the message acts or to which the message is "sent." `Msg` (with a capital M, differentiating it from the key *word* `msg`) is a built-in sort specified by Full Maude for `omods` and object-oriented programming. So, if we want a message `sit@`, which changes a table from unoccupied to occupied, we write:

```
msg sit@ : Oid -> Msg .
```

Note that one feeds sorts, *not classes*, into a message. We identify all objects by their `Oid`s, so feeding in an entire class is unnecessary and extraneous. And the reason for the built-in sort `Msg`? This gets complicated. Messages are like bigger version of operations, but we don't plug an entire object into a message like we plug a variable into an operation. So instead of writing `sit@(< A : TABLE | occupied : O , chairs : N >)`, we write `sit@(A)` *next to* `< A : TABLE | occupied : O , chairs : N >`. Imagine the Maude interpreter is Alice in Wonderland, and she comes across 1) a vial of liquid, and 2) a note that says "Drink Me." So, of course, she drinks the vial. Messages work the exact same way: they are a note sitting next to the object they affect, telling the interpreter to perform some action. We'll later use rewrite laws to define exactly how to perform that action. In a sense, messages are just the first "half" of the process. Is this confusing? Absolutely. We'll clear this all up soon enough. Is the difference between `msg` and `Msg` especially confusing? Maddeningly. The only solace I can offer is that key words are always lower case in Maude, so `msg` must be the key word and `Msg` the built-in sort.

4.5 Rewrite Laws in Object-Oriented Modules

When we last left rewrite laws in Chapter 3, in system modules, we had resolved that they took some state and mapped a transition to another state. In our toy crane example, several of our rewrite laws mapped a concatenation of `State`s, which is a `State` itself, and mapped a transition to another concatenation of `State`s. In other words,

```
rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
```

in terms of the sort `State`, looks like

```
State State State => State State
```

which is really just a specific example of

```
State => State
```

or something like that.

Now, a similar arrangement occurs in object-oriented modules, it just looks different. Messages and objects are both specific types of states. So what a rewrite law looks like in `omods` is:

```
Message Object Object => Object Object
```

Not to say, of course, that a rewrite law in an `omod` is always 1 message + 2 objects = 2 objects; the following are also common patterns:

```
Message Message Object => Object
Message Object => Object Message
```

My point is that objects and messages are just specific states, and any concatenation of messages and objects can be mapped to any other concatenation of messages and objects with a rewrite law, just as concatenations of states were mapped to each other in system modules. How does this specification work? Reflect over the following:

```
subsorts Message Object < State .
```

Of course, the actual Full Maude signature uses different names and words etc., but this explains why we need the built-in sort `Msg`. The message declaration gathers all the required sorts into one state, `Msg`. Then the rewrite law provides the actual transition, using all the information contained in the `Msg` in conjunction with the provided objects.

So all the components of object-oriented rewrite laws, including messages and the new `Msg` sort and the bizarre new syntax of objects, are really just specific examples of everyday rewriting logic; they just imitate Core Maude syntax, and a bit more. In Chapter 3, we created a toy crane environment using rewriting logic. Well, Full Maude simply creates an object-oriented programming environment using rewriting logic. How thoughtful, no?

All right, that was an enormous amount of theory there. Let's look at some examples. First, let's expand our `TABLE` example into a full-fleshed module. We already have a class and a subclass, attributes for both, and a message `sit@`. Let's add a message `borrowchairfrom`, which takes a chair from one table and adds it to another, a message `changetables`, which changes one occupied table to unoccupied and one unoccupied table to occupied, and a message `heaterswitch` that toggles the `next2heater` attribute of `OutDoorTABLE`s.

```
(omod RESTAURANT is
  protecting NAT .
  class TABLE | occupied : Bool , chairs : Nat .
  class OutDoorTABLE | next2heater : Bool .
  subclass OutDoorTABLE < TABLE .
  msgs sit@ heaterswitch : Oid -> Msg .
  msgs _borrowchairfrom_ changetables : Oid Oid -> Msg .

  vars A B : Oid .
  vars M N : Nat .

  rl [sit] : sit@(A) < A : TABLE | occupied : false >
                => < A : TABLE | occupied : true >
  rl [switchon] : heaterswitch(A)
                < A : TABLE | Next2heater : false >
           => < A : TABLE | Next2heater : true > .
  rl [switchoff] : heaterswitch(A)
```

```
                    < A : TABLE | next2heater : true >
                 => < A : TABLE | next2heater : false > .
   rl [move] : changetables(A, B)
                    < A : TABLE | occupied : true >
                    < B : TABLE | occupied : false >
              =>    < A : TABLE | occupied : false >
                    < B : TABLE | occupied : true > .
   rl [takechair] : (A borrowchairfrom B)
                    < A : TABLE | chairs : M >
                    < B : TABLE | chairs : s N >
              =>    < A : TABLE | chairs : s M >
                    < B : TABLE | chairs : N > .
endom)
```

Note that, because object-oriented modules are part of Full Maude and not Core Maude, we have to enclose the `omod` definition in parentheses. Also remember that, even when using mixfix notation, we feed in only `Oid`s, not full obejcts. That is, we wrote

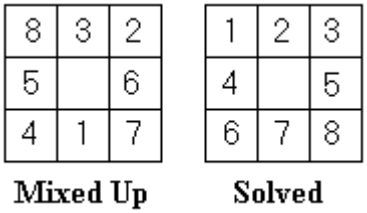
```
(A borrowchairfrom B)
```

instead of

```
< A : TABLE > borrowchairfrom < B : TABLE >
```

Please keep in mind that you are *not* writing in Core Maude, so parsing tricks you might have gotten away with in Core Maude will very easily crash your program in Full Maude. For example, if I did not put parentheses around `A borrowchairfrom B`, Full Maude would understand me to mean `A borrowchairfrom (B < A : TABLE | chairs ... >)`, which of course makes no sense. So, be sure to practice "safe parsing" when programming in Full Maude: use whitespace and parentheses *always*. Finally, when calling an object with multiple attributes in a rewrite law, it is only necessary to state explicitly the attribute that matters for the law. For example, we didn't have to provide any information about the attribute `occupied` in the rewrite law `takechair`.

My favorite `omod` implementation example is the sliding tile puzzle. Imagine (or, just look at the adjacent figure) a three by three grid, with eight numbered tiles free to slide up, down, left, or right into the empty space. So, in our module, we'll have a class `Tile` with an attribute `val`. As for coordinates, we'll create a constructor operation to turn coordinates into an `Oid`.




```
(omod TILE-PUZZLE is
   sorts Value Coord .
   ops One Two Three Four Five Six Seven Eight : -> Value .
   op empty : -> Value .
   ops 0 1 2 : -> Coord .
   op s_ p_ : Coord -> Coord .
```

```
    op `(_`,_`) : Coord Coord -> Oid [ctor] .
    eq s 0 = 1 .
    eq s 1 = 2 .
    eq p 1 = 0 .
    eq p 2 = 1 .

    class Tile | val : Value .
    msg move : Oid Oid -> Msg .
    vars R1 R2 C1 C2 : Coord .       var V : Value .

    crl [l] : move((R1, C1), (R1, C2))
              < (R1, C1) | val : V > < (R1 , C2) | val : empty >
         =>   < (R1 , C2) | val : V > < (R1 , C1) | val : empty >
                                              if C2 == p C1 .

    crl [r] : move((R1, C1), (R1, C2))
              < (R1, C1) | val : V > < (R1, C2) | val : empty >
         =>   < (R1, C2) | val : V > < (R1, C1) | val : empty >
                                              if C2 == s C1 .

    crl [u] : move((R1, C1), (R2, C1))
              < (R1, C1) | val : V > < (R2, C1) | val : empty >
         =>   < (R2, C1) | val : V > < (R1, C1) | val : empty >
                                              if R2 == p R1 .

    crl [d] : move((R1, C1), (R2, C1))
              < (R1, C1) | val : V > < (R2, C1) | val : empty >
         =>   < (R2, C1) | val : V > < (R1, C1) | val : empty >
                                              if R2 == s R1 .
endom)
```

Note that the `Oid`, in this case, was not a typical variable name or word, but rather the result of a constructor. Also note that, for that constructor, we had to use backquotes in front of the parentheses and commas. Since `omod`s are part of Full Maude, in which parentheses and commas are key symbols, we have to put the backquote to use them as normal characters.

So we've wrapped up our exploration of the three types of modules: functional, system, and object-oriented. More importantly, we've seen how Full Maude extends the tools and language of Core Maude, and experienced a bit of the madness inherent in Full Maude programming. In the next two chapters, we'll learn about how to coordinate and manipulate multiple modules in two complex and useful ways, through parameterization and through metaprogramming.

EXERCISES FOR CHAPTER 4

1. Full Maude is an _____ of Core Maude, mimicking the Maude syntax within its own modules but defining a few extra capabilities, most notably _____ modules, declared with the keyword _____ at the beginning and _____ at the end, and _____, the subject of Chapter 5. All modules and commands to be loaded into Full Maude must be enclosed in _____, because Full Maude is written using the input-output capabilities of the provided module _____.

2. The following lines are from different object-oriented modules, and none of them parse correctly. Write the appropriate revision to each statement or declaration that will fix the error(s).

   a) `Msgs debit credit : Oid Nat -> msg .`

   b)
```
rl [switchnums] : A switch B
                  < A : NUMOBJ | n : N >
                  < B : NUMOBJ | n : M >
               =>   < A : NUMOBJ | n : M >
                    < B : NUMOBJ | n : M > .
```

   c) `object PERSON | age : Nat  sign : Zodiac .`

d)
```
rl [older] : ++age(A) < A | age : N , sign : Z >
                    => < A | age : s N , sign : Z > .
```

3. The following object-oriented module defines a basketball scoresheet model, where each player, identified by their jersey number, has a line with their name, scoring record (how many three-pointers, two-pointers, and free throws they've attempted and made) and the number of fouls they've accrued. Examine the module below and answer the questions that follow. You can assume that `throw` is defined somewhere else such that `true` and `false` are equally probable results.

```
1    (omod BASKETBALL-SCORESHEET is
2      pr NAT .       pr STRING .
3      sorts History JerseyNumber .
4      ..................................................................................... .
5
6      op _for_ : Nat Nat -> History [ctor] .
7      op throw : Oid -> Bool .
8
9      class LINE | name : String , threes : History ,
10          twos : History , fts : History ,
11          fouls : Nat .
12
13     msgs ..................................... : Oid Bool -> Msg .
14     msg foul : Oid -> Msg .
15     msg _fouled_ : Oid Oid -> Msg .
16
```

```
17      vars J K : JerseyNumber .      vars X Y N : Nat .
18      var B : Bool .                 var S : String .
19
20      rl [f] :  foul(J)
21                < J : LINE | name : S , fouls : N >
22           =>   < J : LINE | name : S , fouls : s N > .
23      rl [3] : 3A(J, B) < J : LINE | threes : X for Y >
24                     => < J : LINE | threes : if B
25                                     then (s X) for (s Y)
26                                     else X for (s Y) fi > .
27      rl [2] : 2A(J, B) < J : LINE | twos : X for Y >
28                     => < J : LINE | twos : if B
29                                     then (s X) for (s Y)
30                                     else X for (s Y) fi > .
31      rl [ft] : ftA(J, B) < J : LINE | fts : X for Y >
32                     => < J : LINE | fts : if B
33                                     then (s X) for (s Y)
34                                     else X for (s Y) fi > .
35      rl [foulcalled] : (J fouled K)
36                     < J : LINE | > < K : LINE | >
37                =>   foul(J) < J : LINE | >
38                     ftA(K, throw(K)) fta(K, throw(K))
39                     < K : LINE | > .
40      endom)
```

a) What declaration has to go on line 4 so that `JerseyNumber`, which is really just a `Nat`, counts as an object identifier?

b) Fill in the dotted blank on line 13: _____

c) On lines 20-22, the message `foul` is defined using a rewrite rule.

   i. Is the attribute `name` necessary to include in the rule? _____

   ii. Is the attribute `fouls` necessary to include in the rule? _____

d) The following was entered into the Maude environment after having loaded the above module, except the messages have been deleted from the input. What three messages must have occupied the dotted blank to produce the following result?

```
Maude> (rew < 8 : LINE | name : "Kobe" , fouls : 1,
     threes : 0 for 0, twos : 6 for 8, fts : 4 for 4 >
            < 34 : LINE | name : "Shaq" , fouls : 3,
     threes : 0 for 0, twos : 9 for 9, fts : 0 for 8 >
            < 3 : LINE | name : "Iverson" , fouls : 2,
     threes : 2 for 3, twos : 3 for 4, fts : 2 for 4 >
   ..................................................................................... .)

rewrites: 1503 in 20ms cpu (20ms real) (75650 rewr/s)
result Configuration :
     < 3 : LINE | fouls : 3,fts : 2 for 4,twos : 3 for
     5,threes : 2 for 3,name : "Iverson" > < 8 : LINE |
     fouls : 1,fts 4 for 4,twos : 6 for 8,threes : 1 for
     1,name : "Kobe" > < 34 : LINE | fouls : 3,fts : 1 for
     10,twos : 9 for 9,threes : 0 for 0,name : "Shaq" >
```

5. PARAMETERIZATION

5.1 Ideology

In Alvin Toffler's 1970 psycho-social analysis tome Future Shock, the author describes an apartment building made up of a permanent building frame, while steel, modular apartments are hoisted by crane and "plugged into" the frame. This design reflects the general architecture of parameterization, consisting of a frame sort, and a "plugged in" sort. Or, consider a normal electrical outlet. The outlet is only a single structure, by itself not very powerful. But thousands of different appliances (which are not very powerful without an outlet) can be plugged in, creating a combination that is useful. This is the ideology of parameterization. One creates a *parameterized* sort, the outlet, and a *parameter* sort, which plugs into the parameterized sort and creates a useful combination.

For example, say you wanted to make a roster of all the people in a particular school. This would mean a list of teachers, a list of administrators, and a list of students. You could just make a sort `NameList`, to include all three types of names:

```
sorts NameList Name .
subsort Name < NameList .
op __ : NameList NameList -> NameList .
```

But this means that there is no difference in how the module treats teachers, students, and administrators. We could create separate lists for all of them – but this becomes wasteful after a while, when we have the alternative of creating a general sort `Roster`, and parameterizing it. Then we can create three different types of `Rosters`: `Roster(teacher)`, `Roster(student)`, `Roster(administrator)`. As you might expect, parameterization is a very powerful but also rather complex tool, and requires a fair amount of special syntax to learn before we can use it to its fullest extent.

5.2 Syntax of Parameterization

5.2.1 : The Theory

The first requirement for a parameterized anything is a *theory*. A theory sets down the rules for a parameter, a sort of schematic, which the parameter must fulfill in order to be accepted by a parameterized sort. The most common theory, `TRIV`, is, well, trivial:

```
(fth TRIV is
   sort Elt .
endfth)
```

This theory is automatically included in Full Maude, and is the basic example of a theory whose sole requirement is that some sort be present in the parameter. This theory would work well for general parameterized lists. A general list only requires that its elements exist, which is essentially the requirement of `TRIV`. Note that the sort `Elt` is the common abbreviation for

"element" (as in an element of a list or set). Also note that this theory is a *functional theory*: this theory is for the benefit of functional modules. This doesn't mean that a system module or an object-oriented module can't make use of this theory, but rather that there is nothing in the theory which limits it to these modules, such as, say, any requirement about rewrite laws or classes. Finally, note that the theory is in parentheses, because parameterization is a part of Full Maude (see Chapter 4.2).

There are certainly times when a more specific theory is useful. Let's say that you were asked to model (1) a highway map of California, (2) an electrical power grid, and (3) a cross-referenced encyclopedia. What these three examples have in common is that they are a *network*, and can be modeled using a graph of *nodes* and *edges*. The nodes are the (1) cities, (2) houses and power plants, or (3) encyclopedia entries, while the edges are the (1) highways, (2) power lines, or (3) cross-references. So let us create a functional theory `GRAPH` that specifies the requirements of a network:

```
(fth GRAPH is
   sorts Node Edge .
   ops node1 node2 : Edge -> Node .
endfth)
```

Note that we don't specify what the operations `node1` and `node2` actually mean for the sorts. Depending on whether our edges are one-way only, `node1` and `node2` could mean "source" and "target," or they could just be two nodes connected by the edge. All that the theory requires is that there be two operators that return the nodes associated with an edge.

5.2.2 : The Parameterized Module

The next step of parameterization is the actual parameterized module. We declare this module with a special, clunky notation:

```
(fmod EXAMPLE(X :: TRIV) is … endfm)
```

The syntax may look strange, but every bit of it is important. The `(X :: TRIV)` phrase declares that there will be one or more parameterized sorts in the module (all of which will in some way carry around the label `X`) whose parameter must fulfill the theory `TRIV`. Why `X`? Why not. You can use any letter or identifier you wish, as long as you're consistent. But the convention is a single capital letter, usually `X`.

Then, you make sure that anything within the module that relies on a parameter carries that label (we'll use `X` throughout the rest of the chapter). The parameterized sort or sorts carries a parenthesized `X` whenever it appears, while the sorts defined in the theory carry an "`X@`" wherever they go. Thankfully, the operations defined in the theory do *not* carry the `X` with them. Returning to our network example, we write the following module:

```
(fmod NETWORK(X :: GRAPH) is
   sorts NodeSet(X) EdgeSet(X) Network .
```

```
    subsort X@Node < NodeSet(X) .
    subsort X@Edge < EdgeSet(X) .
    op _net_ : NodeSet(X) EdgeSet(X) -> Network [ctor] .
    op __ : NodeSet(X) NodeSet(X) -> NodeSet(X)
                                        [ctor assoc comm id: nnil] .
    op __ : EdgeSet(X) EdgeSet(X) -> EdgeSet(X)
                                        [ctor assoc comm id: nil] .
    op nnil : -> NodeSet(X) [ctor] .
    op nil : -> EdgeSet(X) [ctor] .

    op _in_ : X@Node NodeSet(X) -> Bool .
    op _in_ : X@Edge EdgeSet(X) -> Bool .

    var N : X@Node .var NS : NodeSet(X) .
    var E : X@Edge .var ES : EdgeSet(X) .

    eq N in N NS = true .
    eq N in NS = false [owise] .
    eq E in E ES = true .
    eq E in ES = false [owise] .
endfm)
```

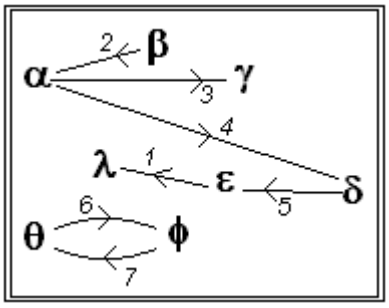
So there's a lot to look at in this module. First, one may notice that `X@Node` and `X@Edge` don't appear to have been initiated as sorts, but in fact they come from the `Node` and `Edge` defined in `GRAPH`. Also note that our two parameterized sorts need to carry that `(X)` with them at all times. Finally, I've included the `_in_` operator, which checks if a `Node` or `Edge` is part of a given corresponding set, just so we have something to do with this example (a common exercise with networks is a search function, which searches for a path between two nodes... the implementation of this, however, is too tangential, and too complex, for the chapter, but is provided in the Examples Appendix).

5.2.3 : The Parameter(s)

Overall, parameterization is really pretty simple: it's just a matter of getting used to and memorizing the syntax involved. And on that note, let's look at some more syntax. We've got the theory, we've got the parameterized module, now let's look at the next requirement: a module that defines the parameter. With respect to our network example, we'll create a rather arbitrary parameter, a graph where the nodes are Greek letters and the edges are natural numbers:

```
(fmod GREEK-GRAPH is
    pr NAT .
    sort Letter .
    ops alpha beta gamma delta epsilon lambda theta phi : -> Letter .
    ops source target : Nat -> Letter .

    eq source(1) = epsilon . eq target(1) = lambda .
    eq source(2) = beta .    eq target(2) = alpha .
    eq source(3) = alpha .   eq target(3) = gamma .
```

```
   eq source(4) = alpha .    eq target(4) = delta .
   eq source(5) = delta .    eq target(5) = epsilon .
   eq source(6) = theta .    eq target(6) = phi .
   eq source(7) = phi .      eq target(7) = theta .
endfm)
```



Of course this is a bit simplistic, but it suffices. The module basically describes the graph to the left. Note that we *must* initiate the above sorts and operations in a module separate from the parameterized module, because we will "plug it into" the `NETWORK` module in a moment, with the help of a *view*.

5.2.4 : Views

So far, in our continuing mission to create an example of parameterization, we've 1) defined the rules for the parameter, 2) declared and defined the parameterized sort, and 3) declared and defined the parameter. As it stands, however, there is no link between the parameter and its rules. After all, how is the Maude compiler supposed to realize that the sorts `Letter` and `Nat` are meant to be the "`X@Node`" and "`X@Edge`," respectively? Why not the other way around? And what about those `node1` and `node2` operators? It probably comes as no surprise that the answer lies in views, the subject of this section.

A view maps the sorts and operations defined in the theory to those in a given parameter. This establishes a link between the two, which allows the compiler to understand the parameter sorts as parameters. We create a view using the special syntax "`view ViewName from THEORY to MODULE is ... endv`." The name doesn't have to be anything special, but it's a <u>good</u> idea to give the view a name related to the problem at hand, and a <u>bad</u> idea to give it the same name as the module or theory. The theory and module names, obviously, are the names of the theory and module to be linked within the view. Remember to enclose the view in parentheses. The actual mapping occurs within the view, using the unexciting but essential key word `to`. This key word allows us to map both sorts and operations in a theory to the corresponding sorts and operations in a module. Returning to our network example:

```
(view Graph-GreekGraph from GRAPH to GREEK-GRAPH is
   sort Node to Letter .
   sort Edge to Nat .
   op node1 to source .
   op node2 to target .
endv)
```

Notice that we still must include the `sort` and `op` key words to denote exactly what we are mapping. Also note that the theory's element always comes before the module's element. Views may also map operators to expressions, though this might prove a tad more complicated. We use the construct `op ..... to term .....` For example, if we had a theory `PEANO` with sort `Num` that requires some starting point (like zero) and some sort of successor operator,

```
(fth PEANO is
   sort Num .
   op 0 : -> Num .  op s_ : Num -> Num .
endfth)
```

and a parameter `THREE-NAT` where all the elements are natural multiples of three,

```
(fmod THREE-NAT is
   pr NAT .
   sort 3Nat .      subsort 3Nat < Nat .
   var N : Nat .
   cmb N : 3Nat if N rem 3 == 0 .
endfm)
```

we could write as our view:

```
(view Peano-3Nat from PEANO to THREE-NAT is
   sort Num to 3Nat .
   var X : Num .
   op 0 to 0 . op s X to term X + 3 .
endv)
```

So, the final step is the one where we actually connect everything: the theory, the parameterized module, the parameter, and the view. We do this in either of two ways: in the Maude environment or within another module. In the Maude environment, all we have to do is attach the view we want in parentheses to the parameterized module. Thus, the two commands below will work:

```
Maude> ( select NETWORK(Graph-GreekGraph) .)
Maude> ( red in NETWORK(Graph-GreekGraph) : lambda in (alpha beta
   lambda phi) .)
```

Recall that all commands in Full Maude must be enclosed by parentheses. If we choose to take the module route, all we need to do is import the parameterized module with the selected view, in the same way as above:

```
(fmod GREEK-NETWORK is
   protecting NETWORK(Graph-GreekGraph) .
   op alltargets : EdgeSet(Graph-GreekGraph) ->
                                      NodeSet(Graph-GreekGraph) .
   op findedge : Letter EdgeSet(Graph-GreekGraph) -> Nat .
   ...
endfm)
```

Anywhere there was an `(X)` in the parameterized module, there's a `(View-Name)` in the instantiation module. The `X@Sort`s have all been replaced by the parameter's sorts that appeared in the view. I have omitted the rest of the operation definitions, because really all you need to see here is an example of how to write the syntax of an *instantiation module*, but feel free to embellish.

So there you have it, the bare minimum of a complete parameterization example: one theory, one parameterized module, one parameter, one view, and some sort of instantiation that makes use of them all. In the next sections, we'll see that we can expand to a much greater extent on this basic design.

5.3 Multiple Parameters

The title says it all: we can parameterize sorts and modules so that they take not just one parameter, but many. The syntax is similar, and only slightly more complicated. When we parameterize a module, we separate those `X :: THEORY` declarations with a vertical bar `|`. Take a look at the following example:

```
(fth OELT is
   sort Elt .
   op _>=_ : Elt Elt -> Bool .
endfth)

(fmod PAIRS(X :: OELT | Y :: TRIV) is
   sorts Pair(X | Y) PairList .
   subsort Pair(X | Y) < PairList .

   op _~_ : X@Elt Y@Elt -> Pair(X | Y) [ctor] .
   op _;_ : PairList PairList -> PairList [ctor assoc id: pnil] .
   op pnil : -> PairList [ctor] .
   op add : Pair(X | Y) PairList -> PairList .

   vars O1 O2 : X@Elt .  vars A1 A2 : Y@Elt .  var P : PairList .

   eq add((O1 ~ A1), (O2 ~ A2) ; P) =  if O1 >= O2
                        then (O2 ~ A2) ; add((O1 ~ A1), P)
                        else (O1 ~ A1) ; (O2 ~ A2) ; P   fi .
   eq add((O1 ~ A1), nil) = O1 ~ A1 .
endfm)
```

The goal in this case is to create a pair of elements where the first element of the pair can be arranged in some order, whatever that may be. Then, we can create a list of the pairs according to the order of their first elements. We use two theories: the first, `OELT`, which describes a sort where order can be determined by the `_>=_` operator. The second, `TRIV`, comes included in Full Maude, and describes any *thing*. When we parameterize the module `PAIRS` according to both of them, we use two different labels that follow their sorts around. Note that since `TRIV` and `OELT` each have a sort `Elt`, the distinction of `X@Elt`, the `Elt` from `OELT`, and `Y@Elt`, the `Elt` from `TRIV`, is rather important. Also note the syntax of `Pair(X | Y)` – again, the vertical bar. If we were to model some sort of triplet, we'd declare the module like

```
(fmod TRIPLE(X :: TRIV | Y :: TRIV | Z :: TRIV) is ...
```

and the parameterized sorts like

```
sort Triple(X | Y | Z) .
```

We can extend the method for as many parameters as we want. When instantiating the parameters, we list the views separated by vertical bars. Say we had the following two views for our parameters:

```
(view OElt-Nat from OELT to NAT is
        sort Elt to Nat .     op _>=_ to _>=_ .     endv)
(view Triv-String from TRIV to STRING is
        sort Elt to String .                  endv)
```

Then we'd connect them all like so:

```
Maude> ( select PAIRS(OElt-Nat | Triv-String) . )
```

5.4 Parameterized Views and Theories

Parameterization in Full Maude is extremely powerful partly because it is extremely flexible. If you can come up with some combination of modules, theories, and views that makes sense, then Maude will allow it. In fact, it will also allow a whole number of combinations that make no sense at all, but these won't run. So we can throw that bare minimum model to the dogs, and, if we want, take a look at some spectacularly weird examples of parameterization.

The game Spider Solitaire requires two decks of cards to play with; let's say that we have one red-backed deck and one blue-backed deck. If we have the module `DECK-COLOR`

```
(fmod DECK-COLOR is
   sort Color .    ops red blue : -> Color .
   op _>=_ : Color Color -> Bool .
   eq red >= blue = true .
endfm)
```

and if we define a deck of cards as a Stack of Cards (the functional module for a stack is given below):

```
(fmod STACK(X :: TRIV) is
   sort Stack(X) .
   subsort X@Elt < Stack(X) .
   op _;_ : Stack(X) Stack(X) -> Stack(X) [ctor assoc id: end] .
   op end : -> Stack(X) [ctor] .
   op top : Stack(X) -> X@Elt .
   op pop : Stack(X) -> Stack(X) .
   op push : X@Elt Stack(X) -> Stack(X) .
   var E : X@Elt . var S : Stack(X) .
   eq top(E ; S) = E .
   eq pop(E ; S) = S .
```

```
    eq push(E, S) = E ; S .
endfm)
```

then we can represent our red and blue decks as `Pair`s with color being the orderable element and a Stack of Cards as the second element. Obviously, we need the following two views:

```
(view Triv-Card from TRIV to CARD-DECK is
        sort Elt to Card .                          endv)
(view OElt-Color from OELT to DECK-COLOR is
        sort Elt to Color .   op _>=_ to _>=_ .     endv)
```

However, to match the `Stack` to the second parameter of `Pair`, we need a parameterized view:

```
(view Triv-Stack(X :: TRIV) from TRIV to STACK(X) is
        sort Elt to Stack(X) .                      endv)
```

And lastly, our `SPIDER-SOLITAIRE` module, which instantiates the whole mish-mash, would start off something like this:

```
(fmod SPIDER-SOLITAIRE is
  pr PAIRS(OElt-Color | Triv-Stack(Triv-Card)) .
  ...
```

In the above declaration, we see exactly why the view needs to be parameterized, because otherwise we would not be able to include the theory `Triv-Card` in the instantiation. One of the most crucial traps of large parameterization systems can be avoided by making sure "all you bases are covered," i.e., that you've connected everything together with views, and all your views are included in the instantiation.

We can also parameterize theories, which makes the syntax even more complex but still doable:

```
(fth FUNCTION(X :: TRIV | Y :: TRIV) is
  op f : X@Elt -> Y@Elt .
endfth)
```

So far, so good, right? The parameterized module, however, is frankly hideous:

```
(fmod FMAP(F :: FUNCTION(X :: TRIV | Y :: TRIV)) is
  sorts IndSet(X) DepSet(Y) .
  subsort X@Elt < IndSet(X) .
  subsort Y@Elt < DepSet(Y) .
  op __ : IndSet(X) IndSet(X) -> IndSet(X)
                                  [ctor assoc comm id: ind-null] .
  op _;_ : DepSet(Y) DepSet(Y) -> DepSet(Y)
                                  [ctor assoc comm id: dep-null] .
  op ind-null : -> IndSet(X) [ctor] .
  op dep-null : -> DepSet(X) [ctor] .
  op fmap : IndSet(X) -> DepSet(Y) .
  var N : X@Elt .  var NS : IndSet(X) .
```

```
   eq fmap(N NS) = f(N) ; fmap(NS) .
   eq fmap(ind-null) = dep-null .
endfm)
```

In the above module, we take our theory of a mapping function and extend it to map one set of elements (the independent variable set) to another (the dependent variable set). The point, though, is to highlight the syntax used: note all the labels and theories included in the declaration of `FMAP`. Try tracing where each operator and sort hail from: whether within the parameterized module itself, or from its various theories.

To illustrate how one might implement a parameterized theory, we use the following parameter:

```
(fmod NEGATIVE is
   pr NAT .   pr INT .
   op _timesneg1 : Nat -> Int .
   var N : Nat .
   eq N timesneg1 = - N .
endfm)
```

This will fit nicely into the `FUNCTION` theory, but we still have to be careful when writing our views. We need to connect `TRIV` to `NAT` and `INT`, first and foremost:

```
(view Triv-Nat from TRIV to NAT is       sort Elt to Nat .     endv)
(view Triv-Int from TRIV to INT is       sort Elt to Int .     endv)
```

and then to connect `FUNCTION` to `NEGATIVE`, we use a parameterized view:

```
(view Fun-Neg(X :: TRIV | Y :: TRIV)
        from FUNCTION(X | Y) to NEGATIVE is
   op f to _timesneg1 .  endv)
```

Notice how we map the parameterized theory: we declare the labels in the view name, and then use them to describe the theory's parameters. Also notice that we don't say anything about sorts `X@Elt` or `Y@Elt`, which we did in the original theory, because the views `Triv-Nat` and `Triv-Int` will take care of those particular connections. We can write something like:

```
Maude> (red in FPAIR(Fun-Neg(Triv-Nat | Triv-Int) :
                                    fmap( 5 9 11 2 0 ) .)
rewrites: 4716 in 100ms cpu (100 ms real) ...
result DepSet`(Triv-Int`) : 0 ; -2 ; -5 ; -9 ; -11
```

Now, at this point, your head might be screaming out dozens of questions like, "Why not use a half-instantiated view like `(view Fun-Neg from FUNCTION(Triv-Nat | Triv-Int) to NEGATIVE)`?" or "can a parameter be parameterized as well?" – and if you've actually tried programming a large parameterization system, then you've probably been frustrated by something that *looks* like it should work but just *doesn't run*. The thing to remember is that all parameterization, and all Full Maude, is ruled by a set of modules written in Core Maude. This means that what we think of "syntax" is really treated like an expression of terms, and as such,

things like error-handling, parsing, and running are all going to be a lot looser, flimsier even, than Core Maude syntax. To make an analogy, `5 * 3 + 2` will appear to the user, almost automatically, as `17`, but if the Maude modules that define the reduction of the expression happen to have given `_+_` a lower `prec` number than `_*_`, the expression will in fact reduce to `25`. Just so, some wild and convoluted declaration of, say, a parameterized view of parameterized theories with multiple parameterized parameters, may look like it's correct to the user, but then Maude will come along and interpret it completely differently. If you want a sure-fire way to get the syntax correct, read the signature of Full Maude. Good luck. Though I cannot give you a full description of the syntax of every combination of parameters and theories and views, I have demonstrated some of the most common combinations above and will list below a few common rules I have learned:

1. When declaring a parameterized module, view, or theory, you list *all* the theories involved and give them labels right away.

√ `(view V(F :: FUNCTION(X :: TRIV | Y :: TRIV) is ...`
X `(view V(F :: FUNCTION) is ...`
X `(view V(F :: FUNCTION(Triv-Nat | Triv-Int) is ...`

2. Once you declare a label for a theory, that label has to follow around consistently every sort or module (not operations, luckily) that you've associated with that theory. Anything you want to depend on a theory, you label.

3. Keep track of your instantiations: make sure the right views go in the right places, separate multiple parameters/views with the vertical bar, make sure you remember which views are parameterized, etc.

Above all, keep trying. If your parameterized system of modules simply crashes, it doesn't mean you don't understand Full Maude, it just means that Full Maude isn't understanding you. Experiment. Be flexible. Let's be honest, parameterization is, at this stage, a few colons and parentheses away from chaotic. *Full Maude will allow you to do almost anything, but this doesn't mean that it will work.*

Connecting and coordinating several modules is no trivial accomplishment. Pretty much every large project will involve many modules that must work together to achieve something meaningful. Parameterization is the most powerful method of connecting these modules from within, from the "inside." In the next chapter, we'll look at how to connect modules from "the outside," some higher level: the meta-level.

EXERCISES FOR CHAPTER 5

1. The bare minimum for a parameterization system is as follows: one _____, to set down the requirements of the parameters, one _____, which connects the _____ to the parameter module, at least two modules (one being the parameter module and the other the _____ module) and some sort of _____ to collect everything together, whether that be within the Maude environment or within another module.

2. Fill in the missing symbols and/or keywords:

   a) `(__________ OElt-Queue(_______ OELT)`
   `from LIST(P) to QUEUE is ...`

   b) `(______ CLASSTHEORY is class ELT . ________)`

   c) `op X < Y _____________ X <= Y and X =/= Y .`

   d) `subsort ____________Elt < List(LABEL) .`
   `***Hint: Elt comes from the theory TRIV`

3. In this question, we'll go step-by-step through your basic, no-tricks parameterization system. The example is very simple: based on a theory that contains a function to *double* something, we'll write a parameterized module that raises something to the 2^nth power.

   a) Write below the functional theory `DOUBLE`, which declares a sort `Something` and an operator `double`.

   b) Write below a parameterized module called `*POWER2`, which, based on the theory `DOUBLE`, declares and defines an operator `*power2`. The `*power2` operator takes in two arguments: the first, a `Something`, and second, a `Nat`.

i. Write the first half of the module, including the module name, the importations, and the operator declaration:

ii. Write the second half of the module, which includes the variable declarations and the equations. We'll define `*power2` recursively so that, for every iteration, it calls `double` on its `Something` argument and decrements its `Nat` argument by 1. Don't forget to include an equation that ends the recursion.

c) Write a view to connect the following module to `DOUBLE`.

```
(fmod INT*2 is
     pr INT .
     op _times2 : Int -> Int .
     eq  N:Int times2 = N:Int * 2 .
endfm)
```

d) Instantiate the above system, with `INT` as your parameter, in two ways:

i. In the Maude environment, using the keyword `select`

```
Maude> _____________________________________
```

ii. In a module `EIGHT`, which declares and defines an operator `*eight`, which multiples an integer by eight (using `*power2`, of course).

6. META-PROGRAMMING

6.1 Ideology

Meta-programming is perhaps one of the most complex and powerful facets of Maude; even though it's not even a part of Core or Full Maude, but rather a standard library module. This separation makes the learning of meta-programming much easier, for we can express and understand the concepts of the meta-level in a language with which we have become quite familiar over the past five chapters. Prepare yourself; this final chapter is twice as long as the others, and with good reason. An appreciation of meta-programming is essential for an appreciation of Maude.

When the prefix "meta" appears in computer programming, it denotes a sense of transcendence or "beyond," and thus metaprogramming extends beyond the ordinary programs with which all previous chapters have concerned themselves. Simply put, a meta-module looks at modules like a programmer, like us – not as a set of executable reduction and rewrite rules, but rather, as a program that can be toyed around with by manipulating its contents. Instead of writing equations and messages on Peano notations and tile-puzzles, we'll be writing equations about the modules `PEANO-NAT` and `TILE-PUZZLE`. In Maude, a meta-level module will govern iconic representations of other Maude modules. Think of a meta-represented module as a sort of voodoo doll, where corresponding parts on doll and human can be maneuvered and affected by a higher, meta-level party. As the chapter develops and actual examples appear, this voodoo mathematics will clarify itself.

In fact, Maude literally clarifies itself through the power of the meta-level, for the language may actually be defined by itself. That is, we can write, in Maude, the module that interprets Maude code. For example, consider the following declaration:

```
op op_:_->_[_] : Qid QidList Qid AttrSet -> OpDecl [ctor] .
```

This is the perfect example of Maude defining itself: the very declaration itself can be understood via the declaration. It is perfect circular logic, which is all right as long as we stay within the circle. The above may be found in the *signature* of Maude, a module that expresses the syntax of Maude... in the syntax of Maude. It's just like a high-school English textbook, which sets down the rules of English, in English. And just like the English language, in Maude we can write about (model) pretty much anything we want, including other Maude modules.

It's a neat trick, but it's also a very powerful tool, as we will see once we understand the syntax necessary to program at the meta-level.

6.2 The Syntax of the Meta-level

As mentioned before, there is a standard library of modules such as `QID`, `STRING`, `NAT`, etc. that accompanies Maude. This library includes the module `META-LEVEL`, a module which defines the meta-representation of functional and system modules (note: object-oriented modules,

because they a component of Full Maude, are not meta-represented in the standard library). `META-LEVEL` imports two other standard modules, `META-TERM` and `META-MODULE`, which, respectively, provide the special syntax specifically for the representation of terms as meta-terms, and modules as meta-modules.

The most important sort defined in `META-TERM` is `Term`. A term is a mathematical concept in set theory (on which Maude heavily relies), but for the purposes of this tutorial one can define a term as an expression formed from the constants and operations of an algebra. Thus, `0` and `s(s(0)) + N:Nat` are examples of terms in `PEANO-NAT-EXTRA`, while `9 of Hearts` and `CardSuit(K of Spades)` are both terms in `CARD-DECK` (see Chapter 2). The `META-LEVEL` module reifies, that is, coalesces, this mathematical concept in the sort `Term`, and its corresponding list sort, `TermList`. Thus, the meta-representation of any term in an algebra will be expressed with respect to `Term` and/or `TermList`. The specifications for these sorts follow any typical list; note particularly the concatenation operation.

```
sorts Term TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [assoc] .
```

The above is a cut-and-paste excerpt from the `META-LEVEL` module; only the relevant information has been quoted. To meta-represent a term, we need to meta-represented the three possible components of term: constants, variables, and operations. We represent a constant as a quoted identifier with its name and sort separated by a period, e.g.: `'0.Nat` or `'K.Number`. Variables are just represented just like constants, except with colons where the period is: `'N:Nat` etc. Finally, stringing all the variables and constants together are the operators, which we express using the operator:

```
op _[_] : Qid TermList -> Term .
```

where the first argument is the name of the operator in quotes, and the second argument is the `TermList` that meta-represents the *operator*'s arguments. So, we meta-represent `K of Spades` as `'_of_[ 'K.Number , 'Spades.Suit ]`, while `s(s(0)) + N:Nat` is `'_+_['s['s['0.Nat]] , 'N:Nat ]`. I have color-coded corresponding elements of the original expression and the meta-representation, since all these quoted identifiers and `TermList`s can get very ugly very fast. I seriously recommend colored pencils and scratch paper when meta-programming.

So that's how we do terms at the meta-level; let's move on now to meta-representing entire modules. A complete functional module necessitates a name, a list of imported modules, a set of sort declarations, a set of subsort declarations, a set of operator declarations, a set of variable declarations (more on this later), a set of membership axioms, and a set of equations. Returning to our very first functional module example, the Peano notation of the natural numbers, we notice that the name is `PEANO-NAT-EXTRA`, there is no import list, there is one sort declaration, there are no subsort declarations, three operator declarations, one variable declaration, and a set of two equations:

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

The `META-MODULE` module defines two very important constructors, which define the meta-representations of functional modules and system modules:

```
op fmod_is_sorts_.____endfm : Qid ImportList SortSet SubsortDeclSet
   OpDeclSet MembAxSet EquationSet -> FModule [ctor] .
op mod_is_sorts_._____endfm : Qid ImportList SortSet SubsortDeclSet
   OpDeclSet MembAxSet EquationSet RuleSet -> Module [ctor] .
```

The first thing that stands out is the special place for sort declarations. Unlike most of the other arguments, which are all sets of declarations made with specially defined keywords, the third arguments takes in a set (constructed with the concatenator `_;_`) of meta-represented sorts, which are just the names of the sorts as quoted identifiers: `'Nat`, `'Bool`, `'State`, etc. So, the sort declaration `sorts Number Suit Card .` would appear as `'Number ; 'Suit ; 'Card` in the third argument of the meta-module constructor.

The second thing that stands out is the lack of a variable declaration set. In fact, there's no syntax to meta-represent a variable declaration. We simply include the variables in the meta-equations with their declared sort as demonstrated earlier, and `META-MODULE` will parse them correctly as declared variables.

As for the rest, I'll write it all down really quickly: the meta-representations of a declaration for equations, importations, operations, rules, etc. all use the same key symbols as in normal modules, save that module names and sorts are all quoted identifiers, and all expressions are meta-represented `Terms`. Subsort declarations must be written one at a time. Rewrite rules do not carry their label in front in brackets, but at the end as an attribute `label('qid)` in brackets. The null constant for the sets is `none` and that for the lists, `nil`. Two examples of modules and their meta-representations are written below:

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

```
fmod 'PEANO-NAT-EXTRA is
  nil
  sorts 'Nat .
  none
  op '0 : nil -> 'Nat [ctor] .
  op 's : nil -> 'Nat [ctor] .
  op '_+_ : 'Nat 'Nat -> 'Nat
                          [none] .
  none
  eq '_+_['N:Nat, '0.Nat] =
                  'N:Nat [none] .
  eq '_+_['s['M:Nat], 'N:Nat] =
       's['_+_['M:Nat, 'N:Nat]]
                          [none] .
endfm
```

```
mod STRING-NAT is
  protecting NAT .
  protecting STRING .

  op n : String -> Nat [ctor] .
  op c : Nat -> String [ctor] .
  var N : Nat .
  var S : String .
  rl [convert] : n(S) =>
                 length(S) .
  crl [back] : c(N) => char(N)
      if N := X:Nat * 2 .
endm
```

```
mod 'STRING-NAT is
  protecting 'NAT .
  protecting 'STRING .
  sorts none .
  none
  op 'n : 'String -> 'Nat [ctor]
.
  op 'c : 'Nat -> 'String [ctor]
.
  none
  none
  rl 'n['S:String] =>
       'length['S:String]
       [label('convert)] .
  crl 'c['N:Nat] => 'char['N:Nat]
       if 'N:Nat :=
            '_*_['X:Nat,
                 's_['s_['0.Nat]]
            ]
       [label('back)] .
endm
```

You might notice that all equations and operations have brackets for a set of attributes, even if that set is `none`. This is required. Also take a look at how the constants are declared in `PEANO-NAT-EXTRA`, with the `nil` where a blank usually would be. This `nil` is an empty `QidList`, which would otherwise list the argument sorts of the operator. Take a look at the conditional rewrite law. Not every condition will involve matching, of course, but I used this example to show that any type of condition: matching, rewrite law, Boolean, etc., will be represented just as it is in a normal module, except with meta-terms instead of expressions. If you're still confused or just curious about how to meta-representations of all the possible variations on modules, either

sit back and study the examples that follow in the chapter, or look up the actual code of META-MODULE in your standard library (prelude.maude).

6.3 The Descent Functions of the META-LEVEL Module

The META-LEVEL module defines three very important and very useful descent functions: metaReduce, metaRewrite, and metaApply. As may be apparent from their names, they simply perform reduction and rewriting of a meta-term, according to the equations and rules in the corresponding meta-module, at the meta-level.

The operation metaReduce takes a Module and Term and returns a ResultPair, which consists of a meta-representation of the answer (as a Term) and the meta-representation of the answer's sort. That is, one enters a certain meta-represented module and a meta-represented term taken from the algebra set down in the module, and metaReduce returns the simplification of the first Term using the rules set down in the (meta-) module. If the module itself (as opposed to its meta-representation) has already been loaded into the Maude environment, the name of the module, quoted and in brackets, may be entered as the Module argument instead of the full-blown meta-representation. So the following two examples are equally valid.

```
Maude> red metaReduce( ['PEANO-NAT-EXTRA],
                       '_+_['s['s['0.Nat]]], '0.Nat] ) .
result ResultPair: {'s['s['0.Nat]], 'Nat}
```

and

```
Maude> red metaReduce( fmod 'PEANO-NAT-MULT is
                         including 'PEANO-NAT-EXTRA .
                         sorts none .
                         none
                         op '_*_ : 'Nat 'Nat -> 'Nat [none].
                         none
                         eq '_*_['N:Nat, '0.Nat] = '0.Nat [none] .
               eq '_*_['N.Nat, 's['M.Nat] ]
                     = '_+_['N.Nat, '_*_['N.Nat,'M.Nat]] [none] .
                         endfm,
                               '_*_['s['0.Nat],'s['s['0.Nat]]] ) .
result  ResultPair: {'s['s['0.Nat]], 'Nat}
```

In the first example, we assumed that we had already loaded the PEANO-NAT-EXTRA module, and indeed we have written it down in full in this chapter. In the second example, we assume that the PEANO-NAT-MULT module has *not* been loaded. Instead of creating a module in the file directory and proceeding as in the first example, we simply included the meta-representation of PEANO-NAT-MULT as the Module argument.

The operation metaRewrite is the counterpart of metaReduce and the meta analog of rewrite. That is, as metaReduce simplifies expressions using equations on the meta-level, metaRewrite transitions states using rewrite laws on the meta-level. The arguments for

metaRewrite are a Module, a Term, and a Bound given either as a Nat or the constant unbounded. The Module and Term arguments follow the same rules as those in metaReduce, and the bound, as in rewrite, limits the number of rewrite laws that can be applied, in case of non-terminating rewrite laws. If one plugs in unbounded for the third argument, the default strategy is applied until termination, when no more rewrite laws can be applied. Needless to say, this is not a good idea for non-terminating rewriting systems. Let's return to the cigarette butts example in Chapter 3. I have changed the number of butts needed to make a cigarette from four butts to two butts.

```
mod CIGARETTES-2 is
   sort State .
   op c : -> State [ctor] .    *** cigarette
   op b : -> State [ctor] .    *** butt
   op __ : State State -> State [ctor] .
   rl [smoke] : c => b .
   rl [makenew]    : b b => c .
endm
```

Assuming that I've already loaded CIGARETTES-2 into the Maude environment, the following two examples hold true:

```
Maude>  red metaRewrite( ['CIGARETTES-2],
   '__['b.State, '__['b.State, '__['b.State, 'b.State] ] ], 10 ) .
result  ResultPair: {'b.State, 'State}
```

and

```
Maude>  red meta-rewrite( ['CIGARETTES-2],
   '__['b.State, '__['b.State, '__['b.State, 'b.State] ] ], 3) .
result  ResultPair: {'__['c.State, 'b.State], 'State}
```

In the first example, six uses of the rewrite laws occur: first, two of makenew, then two of smoke, then one more makenew, and one more smoke. The Bound argument, in the first example, allows for ten applications of rewrite laws, so all is well. However, the second example, almost identical to the first, allows for only three uses of rewrite laws to occur. Thus, the transitions terminate at the state shown.

There exists a third, very important descent function, metaApply, provided by the META-LEVEL module. However, as its uses are too advanced for this section, we will leave the discussion of this function until the end of the chapter. There are more descent functions, most notably metaFrewrite (which meta-rewrites using the frewrite strategy) and metaSearch, which searches at the meta-level (unfortunately, it only confirms whether or not one state can reach another; it does not list the path). These descent functions are powerful but too complicated in their application to explore in this basic tutorial. We will, however, return to metaApply.

6.4 The Cosmetic Functions of Metaprogramming

As one might notice, the meta-representation of terms and modules become increasingly tedious and complicated as the terms and modules become more complex. Thankfully, Full Maude has provided two functions that make meta-representation and its opposite much simpler. That is, the functions *change reflection levels* very easily. When we express a module at the meta-level, or translate a meta-expression back down to normality, we change the level of reflection, that is, the level of meta. Similarly, changing between the meta-level and the meta-meta-level is a matter of reflection levels, and so on. To return to the point at hand, these two functions, `up` and `down`, make the task of metaprogramming much smoother.

The `up` function takes a module and a term as its arguments and returns the meta-representation of the term. Note the difference between `Module` and module, and between `Term` and term. The `up` arguments are not `Qid`s, nor `Term`s formed with the constructors. They are module names and expressions. To demonstrate,

```
Maude>  (red up(PEANO-NAT-EXTRA, s(s(0))) .)
result  Term: 's['s['0.Nat]]
```

This certainly smoothes out the descent functions. Compare the tedious

```
Maude> red metaReduce(['PEANO-NAT-EXTRA],
                      '_+_['s['s['0.Nat]], '0.Nat] ) .
result ResultPair: {'s['s['0.Nat]], 'Nat}
```

with the much simpler

```
Maude>  (red metaReduce(['PEANO-NAT-EXTRA], up(PEANO-NAT-EXTRA,
                        s(s(0)) + 0 )) .)
result  ResultPair: {'s['s['0.Nat]], 'Nat}
```

We can also use `up` on a module, to return its meta-representation:

```
Maude> (red in META-LEVEL : up(FM-CARD-DECK) .)

result Fmodule :
   fmod 'CARD-DECK is
        protecting 'BOOL .
        sorts 'Number ; 'Suit ; 'Card .
        none
        op 'A : nil -> 'Number [ctor] .
        op '2 : nil -> 'Number [ctor] .
             etc. etc. etc. ...
        op 'CardSuit : 'Card -> 'Suit [none] .
        none
        eq 'CardNum['_of_['N:Number,'S:Suit]] = 'N:Number [none] .
        eq 'CardSuit['_of_['N:Number,'S:Suit]] = 'S:Suit [none] .
   endfm
```

I decided to skip most of the seventeen constant declarations, each of which must be written one by one, in `FM-CARD-DECK`, which is just `CARD-DECK` from Chapter 2 with parentheses around it so Full Maude can load it. Note that `BOOL` is imported automatically into every module – but you don't need to worry about that when meta-representing modules by hand.

The `up` function is complemented by the `down_:_.` function, which takes a module name and a meta-represented expression, and returns the expression, written one reflective level lower. This makes the our meta-reduce example even more intelligible, if we couple the two cosmetic functions as follows:

```
Maude>  (down PEANO-NAT-EXTRA : getTerm(metaReduce(
                                          ['PEANO-NAT-EXTRA],
                    up( PEANO-NAT-EXTRA, s(s(0)) + 0 ) )) .)
result  Nat: s(s(0))
```

The convenient power of `up` and `down` may be used as a potent method of checking one's work. For example, if I were to double-check my meta-expression of `PEANO-NAT-MULT` in Section 6.3, I might load `PEANO-NAT-MULT` into Full Maude (with parentheses), and write

```
Maude>  ( red up(PEANO-NAT-MULT) . )
```

and check the result against my work. Or, I might copy and paste my work into the `down` function, and check the result against `PEANO-NAT-MULT`.

6.5 The `metaApply`/`metaXapply` Descent Function

The third descent function, `metaApply`, is an extremely complex and extremely powerful operation that deserves its own section, if not chapter. The `metaApply` basically takes a term and a rewrite law in a given module, and then rewrites the term once by applying the specified law. The Maude declaration of the function looks something like:

```
op metaApply : Module Term Qid Substitution Nat -> ResultTriple .
```

The first three arguments are easily understandable. The `Module` is the meta-representation of the module that defines the terms and laws in question, the `Term` is the given term to be rewritten, and the `Qid` is the name of the rewrite law to be applied. The remaining arguments require a bit of a digression to make sense.

Even before we start the digression, let me point out that, for the most part, we'll be using `metaXapply`, which is almost the same as `metaApply` save for one crucial difference: the `Term` argument of `metaApply` must match the left-hand side of the rewrite rule applied, and match it *exactly* (modulo associativity, commutativity, etc.). When we rewrite using `rew` or `metaRewrite`, we need not worry about this, because the default strategy always rewrites with *extension*; that is, if any pieces of the expression match any rewrite law, it will apply that law.

Not so with `metaApply`. To apply with extension, we use `metaXapply`, which takes – are you ready – seven arguments and returns a quadruple:

```
op metaXapply : Module Term Qid Substitution Nat Bound Nat
                                                  -> Result4Tuple .
```

So let's figure out that `Substitution` argument. When a rewrite law is applied, the variables in the law are replaced with certain constants depending on its pre-rewrite state and the inscrutable whims of the Maude environment. Recall the toy crane example back in Chapter 3; for the forgetful, I have reproduced `the ARCADE-CRANE` system module below. For present purposes, however, I have eliminated the crane itself, thereby reducing the number of rewrite rules we need.

```
mod ARCADE-SANS-CRANE is
   protecting QID .
   sorts ToyID State .
   subsort Qid < ToyID .

   op floor : ToyID -> State [ctor] .
   op on : ToyID ToyID -> State [ctor] .
   op clear : ToyID -> State [ctor] .
   op 1 : -> State [ctor] .
   op _&_ : State State -> State [ctor assoc comm id: 1] .
   vars X Y Z : ToyID .

   rl[unstack] : clear(X) & on(X,Y) =>
                              floor(X) &  clear(X) & clear(Y) .
   rl[stack] : floor(X) & clear(X) & clear(Y) =>
                              clear(X) & on(X,Y) .
   rl[move] : clear(X) & clear(Y) & on(X,Z) =>
                              clear(X) & on(X,Y) & clear(Z) .
endm
```

Consider the state represented in Box C below. Given this state, and the rewrite law `stack` (for example), there are several options for the application of the law. One may stack `'dragondude` on `'soccerball`, or on `'mothergoose`; or one may stack `'mothergoose` on `'dragondude` or `'soccerball`. Thus, given simply the command


```
Maude>  (red metaXapply(
      ['ARCADE-SANS-CRANE],
      up(ARCADE-SANS-CRANE,
clear('mothergoose) &
clear('soccerball) &
clear('dragondude) &
floor('mothergoose) &
floor('teddybear) &
floor('dragondude) &
on('soccerball, 'teddybear)),
'stack, none, 0, unbounded, 0) .)
```

– that is, given the meta-representation of Box C, the rewrite law `stack`, and nothing else, there are four different possible ways of applying `stack`. One may substitute either `'mothergoose` or `'dragondude` for the variable `X` in the `stack` definition, and either `'soccerball` or a choice between `'mothergoose` and `'dragondude` (depending on `X`) for the variable `Y`. To allay this ambiguity, `META-LEVEL` provides a sort `Substitution`, defined by the constructors

```
subsort Assignment < Substitution .
op _<-_ : Qid Term -> Assignment [ctor prec 63] .
op none : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
                                    [assoc comm id: none prec 65] .
```

The Substitution arguments of `metaApply` and `metaXapply` allows us to specify (if we want to) any number of substitutions in order to narrow down the options for applying the rule. If we write

```
Maude>  (red metaXapply(['ARCADE-SANS-CRANE],
                        up(ARCADE-SANS-CRANE,
                              clear('mothergoose) &
                              clear('soccerball) &
                              clear('dragondude) &
                              floor('mothergoose) &
                              floor('teddybear) &
                              floor('dragondude) &
                              on('soccerball, 'teddybear)),
          'stack, ('X:ToyID <- ''dragondude.Qid), 0, unbounded, 0) .)
```

then this limits the application of `stack` to two different results, substituting either `'mothergoose` or `'soccerball` for `Y`. It would also be perfectly valid to enter

```
('X:ToyID <- ''dragondude.Qid) ; ('Y:ToyID <- ''mothergoose.Qid)
```

as the `Substitution` argument, which would narrow down the number of application options of `stack` to one.

After all this, the explanation of the last `Nat` argument (for both `metaApply` and `metaXapply`) is rather simple. The apply functions consider the outcomes of each and every substitution; that is, it evaluates *all* application options. It evaluates each application in some order, and returns the result of the first application. However, given a `Nat N` as the fifth/seventh argument, the operation will return the (`N`+1)th application's result. Thus, plugging into `metaXapply` the module `ARCADE-SANS-CRANE`, the meta-representation of the state in Box C, the rewrite law `stack`, with no substitutions, `0` and `unbounded` (more on these soon), and then, say, two, the operation will return the result of the *third* of the four application options listed previously. Note that if one enters the default, zero, as the seventh argument, the result is simply the *first* of the applications, as the default should be. This optional argument is commonly used in recursive loops, as shall be seen later.

So that takes care of the `metaApply` arguments, but what about the fifth and sixth arguments of `metaXapply`? These natural numbers are the lower and upper bounds on how "deep" we look for matches for the left-hand side of our rewrite rule. Since we're rewriting with extension, a match can in all possibility occur nested within several other operators. For example, if we have a module such as the one below:

```
mod ABCDEFG is
   pr NAT .
   sort State .
   subsort Nat < State .
   ops a b c d e f g : State -> State .
   var X : Nat .
   rl [plus1] : f(X) => X + 1 .
endm
```

we can write an expression of states such that the rewrite law applies only to a phrase nested five times over, i.e., `a(b(c(d(e(f(37))))))`. This, using metaXapply, will usually rewrite to `a(b(c(d(e(38)))))`. But if we put an upper limit on how deep we can extend to rewrite,

```
Maude> red metaXapply(['ABCDEFG],
   'a['b['c['d['e['f['s_^37['0.Nat]]]]]]], 'plus1, none, 0, 3, 0) .
result Result4Tuple?: (failure).Result4Tuple?
```

Here, we can only search at most three levels of nesting deep for a match, and since the `f` operator is five levels deep, the function returns a failure, and will do so, for all values of the sixth argument less than five. We usually give unbounded for this argument, so that `metaXapply` may find matches for the left-hand side no matter how deeply nested they are. Similarly, we can limit how "shallow" we start our search with the first argument. Usually, we want to start at the top of a phrase and enter in 0, but sometimes we only want to rewrite the inside. In the above example, if we gave a lower bound greater than five, we'd bypass the `f` state-operator completely and get another failure:

```
Maude> red metaXapply(['ABCDEFG],
   'a['b['c['d['e['f['s_^37['0.Nat]]]]]]], 'plus1, none, 6,
   unbounded, 0) .
result Result4Tuple?: (failure).Result4Tuple?
```

You might have noticed that our arcade-*sans*-crane example involves a lot of nested `'_&_` operators. `metaXapply` counts a string of associative concatenation operators, such as the ampersand in our example, as one level of nesting, so even though our arcade example looks nested up the wazoo, each `ToyID` is nested only *twice*: first, inside the operators `floor`, `clear`, and `on`, and second, inside the concatenation operators `'_&_`.

Finally, one must ask, what exactly the result of these functions is. For `metaApply`, it is a `ResultTriple`, a `META-LEVEL` sort defined by

```
op {_,_,_} : Term Type Substitution -> ResultTriple .
```

The `Term` element of the `ResultTriple` is simple enough: it is the meta-representation of the $(N+1)^{th}$ term resulting from applying the given rewrite law to the given term in the given module with the given substitutions. In other words, it's the resulting term. The `Type` is the meta-representation of the result's sort, just like in the other descent functions. The `Substitution` element is the concatenation of *all* the substitutions used in the application of the rewrite rule. Thus, if the `metaApply` function applied `stack` by stacking `'dragondude` on `'soccerball`, the `Substitution` element of the resulting `ResultTriple` would be

```
'X:ToyID <- 'dragondude.ToyID ; 'Y:ToyID <- 'soccerball.ToyID
```

The result of `metaXapply` is a `Result4Tuple`, which contains all the above and one more argument, the `Context`. When we apply a rewrite law with extension, we're working with just a piece of the expression, and the `Context` returned in our `Result4Tuple` is 'THE REST,' the pieces of the expression that we didn't work with.

```
Maude>  (red metaXapply(['ARCADE-SANS-CRANE], up(ARCADE-SANS-CRANE,
                                    clear('mothergoose) &
                                    clear('soccerball) &
                                    clear('dragondude) &
                                    floor('mothergoose) &
                                    floor('teddybear) &
                                    floor('dragondude)&
                                    on('soccerball, 'teddybear)),
        'stack,
        ('X:ToyID <- ''dragondude.Qid) ;
        ('Y:ToyID <- ''mothergoose.Qid), 0, unbounded, 0) .)

result Result4Tuple: {'_&_['floor[''mothergoose.Qid],
                           'floor[''teddybear.Qid],
                           'on[''dragondude.Qid,
                           ''mothergoose.Qid],
                           'on[''soccerball.Qid, ''teddybear.Qid],
                           'clear[''dragondude.Qid],
                           'clear[''soccerball.Qid]],
   'State,
             'X:ToyID <- ''dragondude.Qid ;
             'Y:ToyID <- ''mothergoose.Qid,
   '_&_['clear[''soccerball.Qid],
        'floor[''mothergoose.Qid],
        'floor[''teddybear.Qid],
        'on[''soccerball.Qid, ''teddybear.Qid], []]}
```

Those last double brackets, which contain nothing, signify the piece of the expression that you used in `metaXapply`. And, of course, Maude won't print it out as nicely as I have. As you can see, and as I have pointed out many times earlier, the `metaApply` and `metaXapply` functions are extremely complex. However, this complexity lends versatility and flexibility, and allows for the development of internal strategies, the subject of this next and last section.

6.6 Internal Strategies

In Chapter 3, I mentioned that the Maude command `rewrite` is the default rewrite strategy provided by the environment. This strategy, which dictates the order in which rewrite laws of a module are to be applied, matters little in its specifics; suffice to say that it is the default, and is often wholly insufficient for the purposes of a given application. `frewrite` is a step up, but still often insufficient as well. For this reason, Maude enables the user to create his/her own strategy from within a module, using meta-level reflection and the descent functions. For example, consider the following set of rewrite laws:

```
rl [A] : a => b .
rl [B] : b => a .
. . .
```

Needless to say, the default strategy would most likely handle this set poorly, and the current state would alternate between `a` and `b` in a non-terminating fashion, until the natural number limit set down by the user expires. However, we can make use of the `META-LEVEL` functions to create our own strategy. For example, we can create an operation to remove an offensive rewrite law and rewrite according to the default strategy, minus the one rewrite law.

```
fmod REWRITE-EXCEPT is
   protecting META-LEVEL .

   op name : Rule -> Qid .
   op rmv : RuleSet Qid -> RuleSet .
   op rewriteExcept : Module Term Qid Nat -> ResultPair .

   vars L Q : Qid .      var I : ImportList .
   var S : SortSet .     var SS : SubsortDeclList .
   var O : OpDeclSet .   var M : MembAxSet .   var E : EquationSet .
   var RS : RuleSet .    var R : Rule .        vars T1 T2 : Term .
   var N : Nat .

   eq name(rl T1 => T2 [label(Q) A:AttrSet] .) = Q .
   eq name(crl T1 => T2 if C:Condition [label(Q) A:AttrSet] .) = Q .

   eq rmv(R RS, L) = if name(R) == L then RS else R rmv(RS, L) fi .
   eq rmv(none , L) = none .

   eq rewriteExcept(mod Q is I sorts S . SS O M E RS endm,
                         T1, L, N ) =
        metaRewrite(mod Q is I sorts S . SS O M E rmv(RS, L) endm,
                         T1, N ) .
endfm
```

The strategy represented in `rewriteExcept` simply creates a module without the given rewrite law, and calls upon the default strategy (in `metaRewrite`) to rewrite the given term according to

the rules of this new module. Going back to the hypothetical rewrite laws `A` and `B` postulated earlier, one may avoid non-terminating rubbish by calling `rewriteExcept` with either `'A` or `'B` plugged in as the label of the rule to be excluded.

We can play around with more than just rewrite rules. There exist in Maude a host of statement attributes, which I left out of Chapter 2 because it's only at the meta-level that they serve any purpose. For example, equations and membership axioms can have names too. Now, why anybody would want to name an equation seems absolutely baffling, except at the meta-level. Consider the following improvement on the `rewriteExcept`, "`sans`," which extracts a list of rules, equations, or membership axioms from a module:

```
fmod SANS is
   protecting META-LEVEL .

   op sans : Module QidList -> Module .
   op rmvM : MembAxSet Qid -> MembAxSet .
   op rmvE : EquationSet Qid -> EquationSet .
   op rmvR : RuleSet Qid -> RuleSet .

   vars N Q : Qid .      var QL : QidList .    var I : ImportList .
   var S : SortSet .     var SS : SubsortDeclSet .
   var O : OpDeclSet .   vars T1 T2 : Term .   var A : AttrSet .
   var M : MembAx .      var MS: MembAxSet .   var X : Sort .
   var E : Equation .    var ES : EquationSet .
   var R : Rule .        var RS : RuleSet .    var C : Condition .

   eq sans(mod N is I sorts S . SS O MS ES RS endm, Q QL) =
            sans(mod N is I sorts S . SS O
                 rmvM(MS, Q) rmvE(ES, Q) rmvR(RS, Q) endm, QL) .
   eq sans(Z:Module, nil) = Z:Module .

   eq rmvM(mb T1 : X [label(Q) A] . MS, Q) = MS .
   eq rmvM(cmb T1 : X if C [label(Q) A] . MS, Q) = MS .
   eq rmvM(M MS, Q) = M rmvM(MS, Q) [owise] .
   eq rmvM(none, Q) = none .

   eq rmvE(eq T1 = T2 [label(Q) A] . ES, Q) = ES .
   eq rmvE(ceq T1 = T2 if C [label(Q) A] . ES, Q) = ES .
   eq rmvE(E ES, Q) = E rmvE(ES, Q) [owise] .
   eq rmvE(none, Q) = none .

   eq rmvR(rl T1 => T2 [label(Q) A] . RS, Q) = RS .
   eq rmvR(crl T1 => T2 if C [label(Q) A] . RS, Q) = RS .
   eq rmvR(R RS, Q) = R rmvR(RS, Q) [owise] .
   eq rmvR(none, Q) = none .
endfm
```

Now, we can reduce or rewrite *sans* (without) any particular rule or equation or membership axiom we choose. This can be especially useful for proving certain things about our modules. If

we wanted to prove that, out of the following equations, only the first two are necessary to define addition:

```
eq [huey] : N + 0 = N .
eq [duey] : N + s(M) = s(N + M) .
eq [louie] : s(N) + s(M) = s(s(N + M)) .
```

we could run the following tests in `SANS`:

```
Maude> red metaReduce(sans(Z, 'huey), '_+_['s_^7['0.Nat],
   's_^11['0.Nat]]) .
Maude> red metaReduce(sans(Z, 'duey), '_+_['s_^7['0.Nat],
   's_^11['0.Nat]]) .
Maude> red metaReduce(sans(Z, 'louie), '_+_['s_^7['0.Nat],
   's_^11['0.Nat]]) .
```

– the only stipulation being, of course, that we replace `Z` with the `Module` containing `huey`, `duey`, and `louie` (note that, in this case, it must be the actual module, not `['Z]`). In any case, the utility of the equation names is visible. But there's another statement attribute, even more useful, called `metadata`, which allows us to attach a sort of comment (as a `String`) to any statement we choose:

```
eq [juliet] : smell(rose(N:Name)) = sweet .
                    [metadata "A rose by any other name..."] .
```

Sure, it's good for a laugh, but can be incredibly powerful at the meta-level. Let's say we wanted to whip up our arcade-crane into shape, so that, at last, we can build towers instead of just throwing everything on the floor. We could use `metadata` to label certain rewrite rules as 'constructive' and 'destructive,' and then define some operator to look for the 'constructive' tag and execute those rewrite laws (using `metaXapply`) with priority.

One can create internal strategies for extremely complex problems, including theorem proving, searches, and optimization problems. Let us consider an optimization problem. Returning one last time to our cigarette-smoking hobo, we add a new qualification: a more expensive brand of cigarette will contain more tobacco. Thus, the hobo will need fewer cigarette butts to make a new cigarette if he buys the expensive brand, and a greater number of cigarette butts to make a new cigarette if he buys the cheap brand. The question therefore posed is: out of the given brands, out of which brand will the hobo get the most tobacco for his money? First, let us set up our new cigarette module.

```
mod CIGARETTES-3 is
   protecting NAT .
   sorts Brand State .

   ops  DROMEDARY OLDPORT SCARBOROUGH-MAN MISSOURI-FATS
              fail : -> Brand [ctor] .
   op c : Brand Nat -> State [ctor] .
   op b : Brand Nat -> State [ctor] .
```

```
    op __ : State State -> State [ctor assoc comm id: 1] .
    op 1 : -> State [ctor] .   op price : Brand -> Nat .

    vars V W X Y Z : Nat .
    var B : Brand .

    eq price(DROMEDARY) = 312 .      eq price(OLDPORT) = 312 .
    eq price(SCARBOROUGH-MAN) = 364 . eq price(MISSOURI-FATS) = 418 .
    eq price(fail) = 0 .

    rl [smoke] : c(B, X) => b(B, X + 1) .

    rl [makenew] :  b(DROMEDARY, V) b(DROMEDARY, W) b(DROMEDARY, X)
                    b(DROMEDARY, Y) b(DROMEDARY, Z) =>
                              c(DROMEDARY, V + W + X + Y + Z) .

    rl [makenew] :  b(OLDPORT, W)   b(OLDPORT, X)   b(OLDPORT, Y)
                    b(OLDPORT, Z) =>
                              c(OLDPORT, W + X + Y + Z) .

    rl [makenew] :  b(SCARBOROUGH-MAN, X) b(SCARBOROUGH-MAN, Y)
                    b(SCARBOROUGH-MAN, Z) =>
                              c(SCARBOROUGH-MAN, X + Y + Z) .

    rl [makenew] :  b(MISSOURI-FATS, Y)   b(MISSOURI-FATS, Z) =>
                              c(MISSOURI-FATS, Y + Z) .
endm
```

This modification of the `COUNTING-CIGARETTES` module in Chapter 3 is pretty straightforward. The only new concept here is the presence of four rewrite rules with the same label. Such an arrangement is perfectly valid, and certainly a convenience in this example. The above rewrite laws state that a new Dromedary cigarette can be made from five Dromedary butts, a new Oldport from four Oldport butts, a new Scarborough Man from three butts, and a new Missouri Fats from two butts. Also, the `price` equations give the price of the *pack* in *cents*. Next, we create a module to describe the exact scenario with which the hobo is faced: whether to buy a pack of Dromedaries, Oldports, Scarboroughs, or Missouri Fats. Hence the module below:

```
mod CIGARETTE-VENDING-MACHINE is
   protecting CIGARETTES-3 .

   sort BrandSet .
   subsort Brand < BrandSet .

   op vendor : BrandSet -> State .
   op null : -> BrandSet [ctor] .
   op _&_ : BrandSet BrandSet -> BrandSet
                                  [ctor assoc comm id: null] .

   var B : Brand .       var BS : BrandSet .
```

```
   rl [buypack] : vendor(B & BS) =>
                  c(B, 0) c(B, 0) c(B, 0) c(B, 0) c(B, 0)
                  c(B, 0) c(B, 0) c(B, 0) c(B, 0) c(B, 0)
                  c(B, 0) c(B, 0) c(B, 0) c(B, 0) c(B, 0)
                  c(B, 0) c(B, 0) c(B, 0) c(B, 0) c(B, 0) .
endm
```

Note that the above rewrite law specifies no particular brand; the idea is to repeat `metaApply` while incrementing the solution number, so that we select a different brand to substitute as `B` for each iteration. We may rewrite each one of these different applications of `buypack` in succession and then figure out the total cent per cigarette value.

```
fmod CIGARETTE-TEST is
   protecting META-LEVEL .
   protecting CIGARETTE-VENDING-MACHINE .

   sorts CigTotal CigTotalSet .
   subsort CigTotal < CigTotalSet .

   op ct : Term Term -> CigTotal [ctor] .
   op __ : CigTotalSet CigTotalSet -> CigTotalSet
                                    [ctor assoc comm id: none] .
   op none : -> CigTotalSet [ctor] .

   op goTest! : -> CigTotalSet .
   op cigApply : Nat -> ResultTriple .
   op cigTest : Nat -> CigTotalSet .

*** auxiliary operations
   op total : ResultTriple -> CigTotal .
   op getbrand : Substitution -> Term .
   op getnum : ResultPair -> Term .

   var T : Term .        var Y : Type .  var SB : Substitution .
   vars Q1 Q2 : Term .   var TL : TermList .
   var M : Module .      var N : Nat .

   eq goTest! = cigTest(0) .
   eq cigApply(N) = metaApply(['CIGARETTE-VENDING-MACHINE],
                        'vendor['_&_['DROMEDARY.Brand,
                                '_&_['OLDPORT.Brand,
                                '_&_['SCARBOROUGH-MAN.Brand,
                                'MISSOURI-FATS.Brand]]]],
                        'buypack, none, N) .

***key function: CigTest.  Note especially the recursive loop.
   eq cigTest(N) = if cigApply(N) == (failure).ResultTriple?
                   then none
                   else  total(cigApply(N)) cigTest(N + 1)
                   fi .
```

```
    eq total({ T , Y , SB }) = ct(getbrand(SB), getTerm(
                                metaReduce(['CIGARETTES-3],
                '_/_[ 'price[getbrand(SB)],
                      getnum(metaRewrite(['CIGARETTES-3], T, 1000))])
                                                          )
                                    ) .

    eq getbrand( (Q1 <- Q2) ; SB ) = if Q1 == 'B:Brand
                                       then Q2 else getbrand(SB) fi .
    eq getbrand( none ) = 'fail.Brand .

    eq getnum({'__['b[ Q1 , Q2 ] , TL ] , Y }) =
                                '_+_[ Q2 , getnum({ TL , Y }) ] .
    eq getnum({'__['b[ Q1 , Q2 ] , T ] , Y }) =
                                '_+_[ Q2 , getnum({ T , Y }) ] .
    eq getnum({('b[ Q1 , Q2 ] , TL ) , Y }) =
                                '_+_[ Q2 , getnum({ TL , Y }) ] .
    eq getnum({'b[ Q1 , Q2 ] , Y }) = Q2 .

    eq ct( 'fail.Brand , T ) = none .
endfm
```

Consider this the "grand finale" of Maude examples. This particular example has a unique intrinsic stipulation: because the auxiliary functions all use descent functions, the equations defining these auxiliaries must all operate at the meta-level. Because `getnum` is fed a `ResultTriple` from `metaRewrite`, it must deal not with `_quo_` and `Nats` but rather with the meta-representations of these terms.

The use of `metaApply` in this example takes center stage because it creates the internal strategy, but `metaRewrite` and `metaReduce` actually calculate the answers. Our strategy is this: the operation `cigTest` creates a recursive loop which applies `'buypack` by substituting a given brand for the variable `B` of `'buypack`, returns the corresponding `ResultTriple`, *then moves on to the next brand* and concatenates the results. This sequential substitution of brands is accomplished by incrementing the fourth argument, the `Nat`, with every recursive iteration. Thus, if we write

```
Maude>  red goTest! .
```

then the end result will list the first `CigTotal` (using the first brand chosen by the Maude default strategy), then the second `CigTotal` (using the second brand), and so forth, until the recursive loop terminates as `metaApply` exhausts the possible substitutions.

So now that we've analyzed this example thoroughly, what is the solution to the hobo's dilemma? My guess would be to buy Oldports, for they tie Dromedaries as the cheapest, yet have more tobacco than Dromedaries. However, when we run the cigarette test, it turns out that the Missouri Fats are the best value in terms of cents per cigarette smoked. True, a pack of Fats costs $1.06 more than Oldports, but from twenty Fats one can smoke a total of thirty-nine cigarettes, while Oldports produce twenty-six smokes out of one pack. The totals are

Dromedaries: 13¢ per smoke	Oldports: 12¢ per smoke
Scarboroughs: 12¢ per smoke	Missouri Fats: 10¢ per smoke

Realize, of course, that the `_quo_` (or rather, meta-`_quo_`) from `NAT` will round down to the nearest integer, but still, the test will show us which is the cheapest. And as you can see above, the hobo would be better off spending the extra dollar for the Missouri Fats.

I'm not keen on extending this gargantuan chapter any more, but I do hate to end the tutorial with a discussion of addictive carcinogens. Recall the warning at the beginning of the chapter, that meta-programming, though very powerful, is also extremely complex. One should do as much as possible to increase the intelligibility of a meta-program module, making use of techniques such as color-coded `Term`s, plenty of auxiliary operations, and so on, so as to avoid the pitfalls that appear so readily in a complex programming scenario. On a more positive note, meta-programming provides solutions to intricate problems and can create extraordinary programs, and is an essential subject for any Maude adept.

It is my hope that this paper has proved an effective introduction to the syntax and design strategies of Maude. This tutorial focuses on getting people to program in Maude effectively, as opposed to other papers that focus on the math behind it. Both facets are equally important: Maude is both a programming language and a mathematical *tour de force*. Now that you have successfully battled through the swamps of source code and syntax, I would urge you curl up in the easy chair with some classic Alonzo Church, or simply turn to the papers of the Maude team (see References) and dive into the math that makes all the above happen.

EXERCISES FOR CHAPTER 6

1. There standard library module _____ sets down the sorts, operators, and rules for metaprogramming. The basic sort _____ can be constructed in three ways using special key symbols: for constants, as `'name__Sort`, variables, as `'Name__Sort`, and operators, `'opname__TermList__` . We also have as a resource the _____ functions, which reduce, rewrite, search, and apply rules, all at the meta-level.

2. Fill in the appropriate sorts in the following meta-programming operator declarations:

```
op fmod_is_sorts_.____endfm :
```
   a) _____ b) _____ c) _____
   d) _____ e) _____
   f) _____ g) _____
```
   -> FModule [ctor] .
op crl_=>_if_[_] :
```
   h) _____ i) _____ j) _____
   k) _____ `-> Rule [ctor] .`
```
op metaXapply :
```
   l) _____ m) _____ n) _____
   o) _____ p) _____ q) _____ r) _____
   -> r) _____

Given the context of these operator declarations, write the corresponding *null* constant or *default* constant for each sort requested below (note: this will not include all the sorts above):

b) _____
c) _____
d) _____
e) _____
f) _____
g) _____
k) _____
o) _____
p) _____
q) _____
r) _____
s) _____

3. The module CELLS below creates an algebra for classifying cells. The operators do not actually change the cell, but rather describe its characteristics: a cell can be eukaryotic or prokaryotic (animal cell or single-celled organism), and it can be a muscle cell, a neuron, a blood cell, a bacteria, or a phagocyte (defense cells which destroy other cells).

```
mod CELLS is
     sort Cell .
     ops eukaryote prokaryote bacteria bloodcell neuron
          phagocyte muscle : Cell -> Cell [ctor] .
     op C : -> Cell .
     op 1 : -> Cell [ctor] .
     op _&_ : Cell Cell -> Cell [ctor assoc comm id: 1] .
     var X : Cell .
     rl [mitosis] : X => X & X [metadata "two's company"] .
endm
```

Study the transcript of a series of metaXapply trials given below, and answer the questions that follow. The first few lines defines a constant function Z as the meta-module 'CELLS (a technique which often proves very helpful).

```
Maude> fmod CELL-TEST is
> inc META-LEVEL .
> op Z : -> Module [memo] .
> eq Z = (mod 'CELLS is
>             nil
>             sorts 'Cell .
>             none
>             op 'eukaryote : 'Cell -> 'Cell [ctor] .
>             op 'prokaryote : 'Cell -> 'Cell [ctor] .
     etc. etc. etc ...
>             op 'C : nil -> 'Cell [none] .
>             op '1 : nil -> 'Cell [ctor] .
>             op '_&_ : 'Cell 'Cell -> 'Cell
```

```
>                          [ctor assoc comm id('1.Cell)] .
>              none
>              none
>              .....(a).....
>         endm) .
> endfm

Maude>
> red metaXapply(Z, 'phagocyte['bloodcell['eukaryote['C.
Cell]]], 'mitosis, none, 0, unbounded, 0) .

rewrites: 3 in 0ms cpu (0ms real) (~rewrites/second)
result Result4Tuple: {'_&_['phagocyte['bloodcell['eukaryote
['C.Cell]]], 'phagocyte['bloodcell['eukaryote['C.Cell]]]],
'Cell,
   'X:Cell <- .....(b).....], []}

Maude>
> red metaXapply(Z, 'phagocyte['bloodcell['eukaryote['C.
Cell]]], 'mitosis, none, .....(c).....) .

rewrites: 3 in 0ms cpu (0ms real) (~rewrites/second)
result Result4Tuple: {'phagocyte['bloodcell['_&_['eukaryote
['C.Cell], 'eukaryote['C.Cell]]]], 'Cell,
   'X:Cell <- 'eukaryote['C.Cell], .....(d).....}
```

a) I've left out the declarations of most of the descriptive operators, because they are almost identical to the two shown. But the dotted blank marked **(a)** holds the meta-representation of the rewrite rule mitosis, which is crucial. Given the original rewrite rule in the module CELLS, write the meta-representation of this rule below, leaving nothing out.

b) Remember, the `Result4Tuple` returned by `metaXapply` includes, in order, the result term, the meta-sort of the result term, the substitution used in the rewrite, and the context. Based on the `Result4Tuple` returned in the first trial, fill in the dotted blank marked **(b)** below:

c) The second trial of `metaXapply` is missing its fifth, sixth, and seventh arguments. Based on the result of this trial, which of the following triplets of numbers could possibly have filled the dotted blank marked **(c)**?

a. `0 , unbounded , 0`
b. `1 , 2 , 1`
c. `2 , unbounded , 1`
d. `1 , 2 , 2`

d) Which of the following replaces the dotted blank marked **(d)**?

a. `'phagocyte[[]]`
b. `'bloodcell[]`
c. `'phagocyte[bloodcell[]]`
d. `'phagocyte[bloodcell[[]]]`

e) What's the result of:
`metaXapply(Z, 'neuron['C.Cell], 'mitosis, 'X:Cell <- 'C.Cell, 0, unbounded, 1)`?_____

4. On Opposite Day, black is white, 'yes' means 'no,' and everything is the exact reverse of what it is. In the module `OPPOSITE-DAY`, we define an operation `opp-day` that takes a system `Module` and reverses all its rewrite laws, so that `a => b` becomes `b => a`.
   a) Assume the following variable declarations:

```
var R : Rule .          var RS : RuleSet .
```

```
vars T1 T2 : Term .    var C : Condition .
var A : AttrSet .
```

Write a set of equations that define the operator `reverse`, which takes a `RuleSet` and returns that `RuleSet` with all the rewrite laws reversed (as seen above). Don't forget to reverse conditional equations as well.

b) Write the equation that defines `opp-day`. Just for practice, declare all your variables on-the-fly.

c) Declare and define an operation `oppRewrite`, which takes a `Module`, `Term`, and `Bound`, and rewrites the `Term` according to the default Maude strategy, except with all the rewrite rules in the `Module` reversed. Declare your variables however you like. Your operator should return a `ResultPair`.

ANSWERS TO CHAPTER EXERCISES

ANSWERS TO CHAPTER 1 EXERCISES

1. The three types of modules in Maude are declared as _fmod_____, ___mod_____, and ___omod_____. They syntactically stand for _functional____ module, _system_____ module, and _object-oriented_____ module, respectively. The only type of module that is not in Core Maude, but is in Full Maude, is the _omod_____.

2. Identify the following operations as prefix, mixfix, or constant:

```
op _^_ : Nat Nat -> Nat .                    __mixfix_____
op smile : -> Expression .                   __constant____
op to_from_ : Name Name -> Valentine .       __mixfix_____
op grab : Thing -> Action .                  __prefix_____
```

3. When defining the division operator for the natural numbers, one must address the problem of dividing by zero. Do the following on the lines below. a) Define two sorts, `Nat` and `NzNat`, which stands for a non-zero natural number. b) Declare `NzNat` a subsort of `Nat`. c) Define a division operation that takes a `Nat` and an `NzNat`, and returns a `Nat`.

   a) `__sorts Nat NzNat .`_____
   b) `__subsort NzNat < Nat .`_____
   c) `__op _/_ : Nat NzNat -> Nat .`_____

4. Say I want to make an operation that takes a person's *name* and *age*, and returns a statement such as "*Johnny* is *3* years-old".

   a) Choose two standard modules you think I should import for this operation:

   ___NAT_____ and ___QID__ (`STRING` is also a right answer)___

   b) Fill in the blanks for the operation declaration with the correct standard sorts:

```
op _is_years-old : _Qid (or String)_  ___NAT_____
                                          -> AgeStatement [ctor] .
```

   c) Assume that this is the only operation that uses the two imported modules. How should I import them: using `including` or `protecting`? `_protecting_____`

5. Let us define two sorts, `Queue` and `Grab-bag`. In a `Queue`, the order of the elements matter, and so we want to exclude commutativity from the list of its properties. In a `Grab-bag`, order doesn't matter, so we want

commutativity as a property. Furthermore, we want an identity property defined for both, with respect to the `nothing` constant. Associativity we definitely want for both.

Below are two constructors, one for `Queue` and another for `Grab-bag`. Fill in the blanks between the brackets, putting in all the syntax necessary to fulfill the above requirements.

```
op __ : Queue Queue -> Queue   [ assoc id: nothing       ] .
op __ : Grab-bag Grab-bag -> Grab-bag
                                    [ assoc comm id: nothing  ] .
```

6. The combined use of subsorts with operation declarations can be very powerful, and consequently rather tricky. By defining an operation for a sort, one defines it for all of its subsorts. One can think of the < symbol as short for "qualifies as a," e.g., `integer` *qualifies as a* `rational`. With this in mind, and given the following sort and operation declarations,

```
sorts Plea Inquiry Assertion Question Speech .
sort LogEntry .
subsorts Inquiry < Question < Plea .
subsorts Assertion Question < Speech .

op respond : Question -> Speech .
op record : Speech Speech -> LogEntry .

ops p1 p2 : -> Plea .      ops i1 i2 : -> Inquiry .
ops a1 a2 : -> Assertion .  ops q1 q2 : -> Question .
ops s1 s2 : -> Speech .
```

mark which expressions will parse *correctly*, i.e., which will follow the rules of the declarations above.

```
__x__    respond(q1)              ___Speech_____________

__x__    record(q1, i1)                ___LogEntry___________

_____    respond(record(q1, i1))  ___no parse_____________

__x__    record(q1, respond(q1))  ___LogEntry___________
```

_____	respond(p1)	___[Speech, Plea]_____
_____	record(p1, p2)	___[LogEntry]_____
__x__	record(respond(i1), s1)	___LogEntry_____
__x__	record(i1, respond(i1))	___LogEntry_____
_____	respond(s1)	___[Speech, Plea]_____
_____	respond(respond(q1))	___[Speech, Plea]_____

In the second blanks, mark which sort or kind will be the result of the expression, or `no parse` for those terms that will simply not parse at all.

ANSWERS TO CHAPTER 2 EXERCISES

1. There are four, actually five basic requirements for an equation. It must begin with the keyword __eq___, it must have the key symbol __=___ somewhere in it, it must have a __period__ at the end, and it must have within it mathematical phrases consisting of variables and __operators___. Finally, for an equation to be worth its weight, the right-hand side of the equation must be ___simpler_____ than the left-hand side.

2. Decide what the following expressions are by writing in the necessary keyword:
   a) __ceq__ `N mod M = (N - M) mod M if M <= N.`
   b) __cmb__ `N mod M : NzNat if N =/= M`
   `/\ (N - M) mod M :: NzNat .`
   c) __eq___ `N mod M = (N - M) mod M .`
   d) __mb___ `s(0) mod M : NzNat .`
   e) __eq___ `N mod M = if M <= N then (N - M) mod M else N fi .`

3. The modulus operator `mod` returns the remainder of an integer division: for example, 11 `mod` 3 = 2, 4 `mod` 3 = 1, and 30 `mod` 8 = 6. Given the following module (for the `NAT-BOOL` and `PEANO-NAT` modules, see 2.2 and 1.4, respectively),

```
1       fmod MODULUS is
2           protecting NAT-BOOL .
```

```
3             sorts NzNat Zero .
4             subsorts NzNat Zero < Nat .
5             op 0 : -> Zero [ctor] .
6             op _-_ : Nat Nat -> Nat [right id: 0].
7             op _mod_ : Nat NzNat -> Nat [.....] .
8             vars X Y : Nat .  var M : NzNat .
9             cmb X : NzNat if X =/= 0 .
10            eq s X - s Y = if s X > s Y
11                              then X - Y else 0 fi .
12            ..........................................
13            endfm
```

a) Which expression from question #2 would best fill in the dotted blank on line (12) if you want to reduce the following statement?

```
red in MODULUS
   ( s s s s s 0 mod s s s 0 ) mod s s 0 .
```

(e)

b) Write the equation that the identity flag on line (6) replaces.

`eq X - 0 = X .`

c) Lines (10) and (11), which, along with line (6), define the subtraction operator, could be replaced by the following two equations:

```
ceq s X - s Y = X - Y if s X > s Y .
eq ..... = 0 [.....] .
```

Choose the expression-and-flag pair that best fills in the two dotted blanks.

a.	`s X - s Y ... memo`	**C.**	`s X - s Y ... owise`
b.	`X - Y ... owise`	d.	`X - Y ... gather (& e)`

d) The constant operator `0` in line (5) is an example of subsort overloading.

e) What flag(s) should fill the dotted blank on line (7) so that 14 `mod` 5 `mod` 3 reduces to `0`?

a.	`assoc`	c.	`gather (E e)`
B.	`gather (e E)`	d.	`gather(& &)`

4. The sort `Deck` is a list of cards (refer to the `CARD-DECK` module in 2.1) defined by the subsort declaration

```
subsort Card < Deck .
```

and the operator `_;_`, defined by

```
op _;_ : Deck Deck -> Deck [ctor assoc id: null] .
op null : -> Card [ctor] .
```

a) Write an equation that defines the operation `topcard`, which takes a `Deck` and returns the top card (i.e., the first element.) Assume that the variable `C` of sort `Card` has already been declared.

```
eq topcard(C ; D:Deck) = C .
```

b) Let sort `FemCard` be defined with the subsort relation `FemCard < Card`. Write one (or more) membership axiom(s), conditional if you wish, that makes a `Card` a `FemCard` if the suit is `Hearts` *or* the card is a Queen. You might want to use the operations `CardNum` and `CardSuit` defined in `CARD-DECK`, but then again you might not. Feel free to declare your variables any way you like.

```
mb N:Number of Hearts : FemCard .
mb Q of S:Suit : FemCard .
```

c) Write one or more equation(s) that define(s) the operator `firstFemCard`, which returns the first `FemCard` drawn from a deck. You *must* use recursion.

i. Write it using the if_then_else_fi operator.

```
var C : Card .          var D : Deck .
eq firstFemCard(C ; D) = if CardNum(C) == Q or
                                CardSuit(C) == Hearts
                            then C
                            else firstFemCard(D) fi .
eq firstFemCard(null) = null .
```

ii. Write it using a conditional equation and the `_::_` operator.

```
ceq firstFemCard(C ; D) = C if C :: FemCard .
eq firstFemCard(C ; D) = firstFemCard(D) [owise] .
eq firstFemCard(null) = null .
```

iii. Write it using a conditional equation and pattern matching.

```
ceq firstFemCard(C ; D) = C if Q of S:Suit := C .
ceq firstFemCard(C ; D) = C if N:Number of Hearts := C .
eq firstFemCard(C ; D) = firstFemCard(D) [owise] .
eq firstFemCard(null) = null .
```

ANSWERS TO CHAPTER 3 EXERCISES

1. While __equations_____ describe rules of simplification (writing the same thing a different way), a _rewrite law__ maps a ___transition___ between two states (a *change* in constitution). And while the right-hand side of a(n) ___equation_____ should be simpler than the left-hand side, this is not necessarily true for the other. This latter uses the key symbol => to emphasize its _irreversibility____, that is, the fact that it only works in one direction.

   Both equations and rewrite laws may be found in a _system_____ module, initiated with the keyword _mod___.

2. The following module sets down the rules for a rewrite game, called "Zip-Zap-Zop." Whenever the constants `zip`, `zap`, and `zop` appear in that order, they become a `win`. And `win`s can change into all sorts of combinations of the three. In addition, four `zop`s in a row ends the game. The goal is, then, to apply the correct rewrite rules so that we end with one `win`. Look over the module and answer the questions that follow:

```
1        mod ZIP-ZAP-ZOP is
2              sort GameState .
3              op _>_ : GameState GameState ->
4                           GameState [ctor assoc id: 1] .
5              op 1 : -> GameState [ctor] .
6              ops zip zap zop : -> GameState [ctor] .
7              ops win gameover : -> GameState [ctor] .
8
9              rl [winner!] : zip > zap > zop => win .
10             rl [scatter1] : win => zop > zip .
11             rl [scatter2] : win => zip > zap .
12             rl [scatter3] : win => zop > zop .
13             rl [gameend] : ..................................................... .
14       endm
```

   a) What's missing from line (4) that would allow Maude to rewrite `zop > zap > zip`?

   ___comm_____

   b) Fill in the dotted blank on line (13):

   __zop > zop > zop > zop => gameover_____

   c) Consider the expression `win zop win`:

      i. Does an unbounded rewrite of this expression always terminate? __yes_____

ii. What are the five rewrites, in order of application, that must occur in order for the above expression to reach a `gameover`? In the first blank, put the label of the rewrite law you apply, and in the second, the state that results.

`win > zop > win`

1. `__scatter3___` => `_win > zop > zop > zop_____`
2. `__scatter2___` => `_zip > zap > zop > zop > zop__`
3. `__winner!____` => `_win > zop > zop_____`
4. `__scatter3___` => `_zop > zop > zop > zop_____`
5. `__gameend_____` => `__gameover_____`

d) Let's say we want to add a rule that will change `zap` into `zip` as long as this immediately produces a `win` in the expression. Write a conditional rewrite law `change` that transforms `zap > S` into `zip > S`, and the condition is another rewrite law. Assume variables `S` and `P` of `GameState` have already been declared.

```
crl [change] : zap > S => zip > S
                              if zip > S => win > P .
```

3. Two separate searches were performed with the complete `ZIP-ZAP-ZOP` module (that is, including the conditional rewrite law `change`), and the transcription appears below. The ellipses appear in the place of unimportant information, so don't worry about those, but some important parts have been substituted by dotted blanks. Study the searches carefully and answer the questions below.

First search:

```
Maude> search zap > zap > zip > zap > zop ....(a).... win .
...

No solution.
states: 22  rewrites: 106 ...
```

Second search:

```
Maude> search zap > zap > zip > zap > zop ....(a).... win .
...

Solution 1 (state 18)
states: 19  rewrites: 103 ...
empty substitution
```

```
No more solutions.
states: 22  rewrites: 106 ...
Maude> show .....(b)..... .
state 0, GameState: zap > zap > zip > zap > zop
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 1, GameState: zap > zap > win
==[ rl win => zop > zop [label scatter3] . ]===>
state 3, GameState: zap > zap > zop > zop
==[ crl ... [label change] . ]===>
state 9, GameState: .....(c).....
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 11, GameState: win > zop
==[ rl .....(d)..... [label .....(d).....] . ]===>
state 16, GameState: zip > zap > zop
==[ rl zip > zap > zop => win [label winner!] . ]===>
state 19, GameState: win
```

a) As you can see, the first and the second searches are almost identical, except for the symbol that goes in the two dotted blanks labeled **(a)**.

   i. What symbol must have been used for the first search? ____=>!_____

   ii. What symbol must have been used for the second search? ___=>+_____

   iii. If we had written `zop > zop > zop` instead of `win` in the first search, would Maude have found a solution? __yes_____

   iv. If we had written `zop > zop > zop` instead of `win` in the *second* search, would Maude have found a solution? __yes_____

b) Fill in the dotted blank labeled **(b)**: ___path 18_____

If you had entered in `show path 11 .` instead, what would the last line of your result be? ___state 11, GameState: win > zop_____

c) Refer to question 2.d) and the rest of the search transcript. How should you fill the dotted blank labeled **(c)**? ___zip > zap > zop > zop _____

d) Based on the evidence of states 11 and 16, what must go in the two dotted blanks labeled **(d)**?

___win => zip > zap_____

and

___scatter2_____

ANSWERS TO CHAPTER 4 EXERCISES

1. Full Maude is an ____extension_____ of Core Maude, mimicking the Maude syntax within its own modules but defining a few extra capabilities, most notably _object-oriented_____ modules, declared with the keyword _omod____ at the beginning and __endom____ at the end, and __parameterization_____, the subject of Chapter 5. All modules and commands to be loaded into Full Maude must be enclosed in ____parentheses_____, because Full Maude is written using the input-output capabilities of the provided module _LOOP-MODE_____.

2. The following lines are from different object-oriented modules, and none of them parse correctly. Write the appropriate revision to each statement or declaration that will fix the error(s).

   a) `Msgs debit credit : Oid Nat -> msg .`

   ```
   msgs debit credit : Oid Nat -> Msg .
   ```

   b)
   ```
   rl [switchnums] : A switch B
                        < A : NUMOBJ | n : N >
                        < B : NUMOBJ | n : M >
                  =>    < A : NUMOBJ | n : M >
                        < B : NUMOBJ | n : M > .
   ```

   ```
   rl [switchnums] : (A switch B)
                        < A : NUMOBJ | n : N >
                        < B : NUMOBJ | n : M >
                  =>    < A : NUMOBJ | n : M >
                        < B : NUMOBJ | n : M > .
   ```

   c) `object PERSON | age : Nat  sign : Zodiac .`

   ```
   class PERSON | age : Nat , sign : Zodiac .
   ```

   d)
   ```
   rl [older] : ++age(A) < A | age : N , sign : Z >
                        => < A | age : s N , sign : Z > .
   ```

```
      rl [older] : ++age(A) < A : PERSON | age N >
                        => < A : PERSON | age s N > .
```

3. The following object-oriented module defines a basketball scoresheet model, where each player, identified by their jersey number, has a line with their name, scoring record (how many three-pointers, two-pointers, and free throws they've attempted and made) and the number of fouls they've accrued. Examine the module below and answer the questions that follow. You can assume that `throw` is defined somewhere else such that `true` and `false` are equally probable results.

```
1     (omod BASKETBALL-SCORESHEET is
2       pr NAT .          pr STRING .
3       sorts History JerseyNumber .
4       ..................................................................................... .
5
6       op _for_ : Nat Nat -> History [ctor] .
7       op throw : Oid -> Bool .
8
9       class LINE | name : String , threes : History ,
10            twos : History , fts : History ,
11            fouls : Nat .
12
13      msgs ..................................... : Oid Bool -> Msg .
14      msg foul : Oid -> Msg .
15      msg _fouled_ : Oid Oid -> Msg .
16
17      vars J K : JerseyNumber .     vars X Y N : Nat .
18      var B : Bool .                      var S : String .
19
20      rl [f] :    foul(J)
21                  < J : LINE | name : S , fouls : N >
22            =>    < J : LINE | name : S , fouls : s N > .
23      rl [3] : 3A(J, B) < J : LINE | threes : X for Y >
24                        => < J : LINE | threes : if B
25                                        then (s X) for (s Y)
26                                        else X for (s Y) fi > .
27      rl [2] : 2A(J, B) < J : LINE | twos : X for Y >
28                        => < J : LINE | twos : if B
29                                        then (s X) for (s Y)
30                                        else X for (s Y) fi > .
```

```
31       rl [ft] : ftA(J, B) < J : LINE | fts : X for Y >
32                         => < J : LINE | fts : if B
33                                         then (s X) for (s Y)
34                                         else X for (s Y) fi > .
35       rl [foulcalled] : (J fouled K)
36                         < J : LINE | > < K : LINE | >
37                    =>   foul(J) < J : LINE | >
38                         ftA(K, throw(K)) fta(K, throw(K))
39                         < K : LINE | > .
40       endom)
```

a) What declaration has to go on line 4 so that `JerseyNumber`, which is really just a `Nat`, counts as an object identifier?

```
subsorts Nat < JerseyNumber < Oid .
```

b) Fill in the dotted blank on line 13: ___3A 2A ftA _____

c) On lines 20-22, the message `foul` is defined using a rewrite rule.
   i. Is the attribute `name` necessary to include in the rule? _____yes_____
   ii. Is the attribute `fouls` necessary to include in the rule? _____no_____

d) The following was entered into the Maude environment after having loaded the above module, except the messages have been deleted from the input. What three messages must have occupied the dotted blank to produce the following result?

```
Maude> (rew < 8 : LINE | name : "Kobe" , fouls : 1,
       threes : 0 for 0, twos : 6 for 8, fts : 4 for 4 >
              < 34 : LINE | name : "Shaq" , fouls : 3,
       threes : 0 for 0, twos : 9 for 9, fts : 0 for 8 >
              < 3 : LINE | name : "Iverson" , fouls : 2,
       threes : 2 for 3, twos : 3 for 4, fts : 2 for 4 >
    ..................................................................................... .)

rewrites: 1503 in 20ms cpu (20ms real) (75650 rewr/s)
result Configuration :
       < 3 : LINE | fouls : 3,fts : 2 for 4,twos : 3 for 5,threes : 2
       for 3,name : "Iverson" > < 8 : LINE | fouls : 1,fts 4 for 4,twos
```

```
: 6 for 8,threes : 1 for 1,name : "Kobe" > < 34 : LINE | fouls :
3,fts : 1 for 10,twos : 9 for 9,threes : 0 for 0,name : "Shaq" >

      (3 fouled 34) 3A(8, true) 2A(3, false)
```

ANSWERS TO CHAPTER 5 EXERCISES

1. The bare minimum for a parameterization system is as follows: one __theory_____, to set down the requirements of the parameters, one __view_____, which connects the __theory_____ to the parameter module, at least two modules (one being the parameter module and the other the __parameterized__ module) and some sort of __instantiation____ to collect everything together, whether that be within the Maude environment or within another module.

2. Fill in the missing symbols and/or keywords:

   a)
   ```
   (__view___ OElt-Queue(_P ::__ OELT)
        from LIST(P) to QUEUE is ...
   ```

   b)
   ```
   (_oth__ CLASSTHEORY is  class ELT .   _endoth_)
   ```

   c)
   ```
   op X < Y __to term_____ X <= Y and X =/= Y .
   ```

   d)
   ```
   subsort __LABEL@Elt < List(LABEL) .
               ***Hint: Elt comes from the theory TRIV
   ```

3. In this question, we'll go step-by-step through your basic, no-tricks parameterization system. The example is very simple: based on a theory that contains a function to *double* something, we'll write a parameterized module that raises something to the 2^nth power.

   a) Write below the functional theory `DOUBLE`, which declares a sort `Something` and an operator `double`.

   ```
   (fth DOUBLE is
      sort Something .
      op double .
   endfth)
   ```

   b) Write below a parameterized module called `*POWER2`, which, based on the theory `DOUBLE`, declares and defines an operator `*power2`. The `*power2` operator takes in two arguments: the first, a `Something`, and second, a `Nat`.

      i. Write the first half of the module, including the module name, the importations, and the operator declaration:

```
(fmod *POWER2(X :: DOUBLE) is
      protecting NAT .
      op _*power2_ : X@Something Nat
                              -> X@Something .
```

ii. Write the second half of the module, which includes the variable declarations and the equations. We'll define `*power2` recursively so that, for every iteration, it calls `double` on its `Something` argument and decrements its `Nat` argument by 1. Don't forget to include an equation that ends the recursion.

```
      var N : Nat .     var S : X@Something .
      eq S *power2 (s N) = double(S) *power2 N .
      eq S *power2 0 = S .
endfm)
```

c) Write a view to connect the following module to `DOUBLE`.

```
(fmod INT*2 is
      pr INT .
      op _times2 : Int -> Int .
      eq  N:Int times2 = N:Int * 2 .
endfm)
```

```
(view Double-Int from DOUBLE to INT*2 is
   sort Something to Int .
   op double to _times2 .
endv)
```

d) Instantiate the above system, with `INT` as your parameter, in two ways:

i. In the Maude environment, using the keyword `select`

```
Maude> (select *POWER2(Double-Int) .)
```

ii. In a module `EIGHT`, which declares and defines an operator `*eight`, which multiples an integer by eight (using `*power2`, of course).

```
(fmod EIGHT is
```

```
        protecting *POWER2(Double-Int) .
        op *eight : Int -> Int .
        eq X:Int *eight = X:Int *power2 3 .
endfm)
```

ANSWERS TO CHAPTER 6 EXERCISES

1. There standard library module ___META-LEVEL_____ sets down the sorts, operators, and rules for metaprogramming. The basic sort ____Term_____ can be constructed in three ways using special key symbols: for constants, as `'name.Sort`, variables, as `'Name:Sort`, and operators, `'opname[TermList]` . We also have as a resource the ____descent_____ functions, which reduce, rewrite, search, and apply rules, all at the meta-level.

2. Fill in the appropriate sorts in the following meta-programming operator declarations:

```
op fmod_is_sorts_.____endfm :
```

a) __Qid__ b) _ImportList_____ c) _SortSet_____

d) __SubsortDeclSet_____ e) __OpDeclSet_____

f) ___MembAxSet_____ g) ___EquationSet_____

```
-> FModule [ctor] .
op crl_=>_if_[_] :
```

h) __Term_____ i) ___Term_____ j) ___Condtion_____

k) __AttrSet_____ `-> Rule [ctor].`

```
op metaXapply :
```

l) ___Module_____ m) ___Term_____ n) __Qid_____

o) __Substitution____ p) __Nat___ q) __Bound___ r) ___Nat___

-> s) ___Result4Tuple_____

Given the context of these operator declarations, write the corresponding *null* constant or *default* constant for each sort requested below (note: this will not include all the sorts above):

b) __nil_____

c) __none_____

d) __none_____

e) __none_____

f) __none_____

g) __none_____

k) __none_____

o) __none_____

p) __0___

q) __unbounded_____

r) __0___

s) `(failure).Result4Tuple?`

3. The module `CELLS` below creates an algebra for classifying cells. The operators do not actually change the cell, but rather describe its characteristics: a cell can be eukaryotic or prokaryotic (animal cell or single-celled organism), and it can be a muscle cell, a neuron, a blood cell, a bacteria, or a phagocyte (defense cells which destroy other cells).

```
mod CELLS is
      sort Cell .
      ops eukaryote prokaryote bacteria bloodcell neuron
            phagocyte muscle : Cell -> Cell [ctor] .
      op C : -> Cell .
      op 1 : -> Cell [ctor] .
      op _&_ : Cell Cell -> Cell [ctor assoc comm id: 1] .
      var X : Cell .
      rl [mitosis] : X => X & X [metadata "two's company"] .
endm
```

Study the transcript of a series of `metaXapply` trials given below, and answer the questions that follow. The first few lines defines a constant function `Z` as the meta-module `'CELLS` (a technique which often proves very helpful).

```
Maude> fmod CELL-TEST is
> inc META-LEVEL .
> op Z : -> Module [memo] .
> eq Z = (mod 'CELLS is
>                 nil
>                 sorts 'Cell .
>                 none
>                 op 'eukaryote : 'Cell -> 'Cell [ctor] .
>                 op 'prokaryote : 'Cell -> 'Cell [ctor] .
      etc. etc. etc ...
>                 op 'C : nil -> 'Cell [none] .
>                 op '1 : nil -> 'Cell [ctor] .
>                 op '_&_ : 'Cell 'Cell -> 'Cell
>                             [ctor assoc comm id('1.Cell)] .
>                 none
>                 none
>                 .....(a).....
>           endm) .
> endfm
```

```
Maude>
> red metaXapply(Z, 'phagocyte['bloodcell['eukaryote['C.
Cell]]], 'mitosis, none, 0, unbounded, 0) .

rewrites: 3 in 0ms cpu (0ms real) (~rewrites/second)
result Result4Tuple: {'_&_['phagocyte['bloodcell['eukaryote
['C.Cell]]], 'phagocyte['bloodcell['eukaryote['C.Cell]]]], 'Cell,
   'X:Cell <- ……(b)……, []}

Maude>
> red metaXapply(Z, 'phagocyte['bloodcell['eukaryote['C.
Cell]]], 'mitosis, none, ……(c)……) .

rewrites: 3 in 0ms cpu (0ms real) (~rewrites/second)
result Result4Tuple: {'phagocyte['bloodcell['_&_['eukaryote
['C.Cell], 'eukaryote['C.Cell]]]], 'Cell,
   'X:Cell <- 'eukaryote['C.Cell], ……(d)……}
```

a) I've left out the declarations of most of the descriptive operators, because they are almost identical to the two shown. But the dotted blank marked **(a)** holds the meta-representation of the rewrite rule mitosis, which is crucial. Given the original rewrite rule in the module `CELLS`, write the meta-representation of this rule below, leaving nothing out.

```
rl 'X:Cell => '_&_['X:Cell, 'X:Cell] [label('mitosis)
                                      metadata("two's company")] .
```

b) Remember, the `Result4Tuple` returned by `metaXapply` includes, in order, the result term, the meta-sort of the result term, the substitution used in the rewrite, and the context. Based on the `Result4Tuple` returned in the first trial, fill in the dotted blank marked **(b)** below:

```
'phagocyte['bloodcell['eukaryote['C.Cell]]]
```

c) The second trial of `metaXapply` is missing its fifth, sixth, and seventh arguments. Based on the result of this trial, which of the following triplets of numbers could possibly have filled the dotted blank marked **(c)**?

a. `0 , unbounded , 0`
c. `2 , unbounded , 1`
B. `1 , 2 , 1`
d. `1 , 2 , 2`

d) Which of the following replaces the dotted blank marked **(d)**?

a. `'phagocyte[[]]`

b. `'bloodcell[]`

c. `'phagocyte[bloodcell[]]`

D. `'phagocyte[bloodcell[[]]]`

e) What's the result of:

```
metaXapply(Z, 'neuron['C.Cell], 'mitosis, 'X:Cell <- 'C.Cell, 0,
unbounded, 1)?___(failure).Result4Tuple?_____
```

4. On Opposite Day, black is white, 'yes' means 'no,' and everything is the exact reverse of what it is. In the module `OPPOSITE-DAY`, we define an operation `opp-day` that takes a system `Module` and reverses all its rewrite laws, so that `a => b` becomes `b => a`.

a) Assume the following variable declarations:

```
var R : Rule .          var RS : RuleSet .
vars T1 T2 : Term .     var C : Condition .
var A : AttrSet .
```

Write a set of equations that define the operator `reverse`, which takes a `RuleSet` and returns that `RuleSet` with all the rewrite laws reversed (as seen above). Don't forget to reverse conditional equations as well.

```
eq reverse(rl T1 => T2 [A] . RS) =
                              rl T2 => T1 [A] . reverse(RS) .
eq reverse(crl T1 => T2 if C [A] . RS) =
                           crl T2 => T1 if C [A] . reverse(RS) .
eq reverse(none) = none .
```

b) Write the equation that defines `opp-day`. Just for practice, declare all your variables on-the-fly.

```
eq opp-day(mod Q:Qid is I:ImportList sorts S:SortSet .
   SS:SubsortDeclSet O:OpDeclSet M:MembAxSet
   E:EquationSet RS:RuleSet endm) =  mod Q:Qid is
                                                I:ImportList
                                                sorts SortSet .
                                                SS:SubsortDeclSet
                                                O:OpDeclSet
                                                M:MembAxSet
```

```
  E:EquationSet
  reverse(RS:RuleSet)
endm .
```

c) Declare and define an operation `oppRewrite`, which takes a `Module`, `Term`, and `Bound`, and rewrites the `Term` according to the default Maude strategy, except with all the rewrite rules in the `Module` reversed. Declare your variables however you like. Your operator should return a `ResultPair`.

```
op oppRewrite : Module Term Bound -> ResultPair .
var M : Module .     var T : Term .          var B : Bound .

eq oppRewrite(M, T, B) =
               metaRewrite(opp-day(M), T, B) .
```

Appendix : Examples

```
*************************************************************************
***  Chapter 1 - Foundation
***    Section 4:  Constructors and Operator Attributes
***
***  PEANO-NAT : sets down the algebra for the Peano notation of the
***  natural numbers, using two constructors - the successor s and 0.
***
*************************************************************************

fmod PEANO-NAT is

  sort Nat .

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

endfm



*************************************************************************
***  Chapter 1 - Foundation
***    Section 4: Constructors and Operator Attributes
***
***  SETS-AND-LISTS: sets down the fundamentals for the Set and List
***  sorts, including the main binary operator attributes "assoc,"
***  "comm," and "id: "
***
*************************************************************************

fmod SETS-AND-LISTS is

  sorts Set List Elt .
  subsort Elt < Set .
  subsort Elt < List .

  op nil : -> List [ctor] .
  op none : -> Set [ctor] .

  op _;_ : List List -> List [ctor assoc id: nil] .
  op __ : Set Set -> Set [ctor assoc comm id: none] .

endfm



*************************************************************************
***  Chapter 1 - Foundation
***    Section 5: Kinds
***
***  ANIMALS : illustrates the use of kinds within connected
***  components using a dog-breeding model.
```

```
***
*****************************************************************

fmod ANIMALS is

  sorts Animal Vegetable Dog .
  sorts Terrier Hound Toy Sporting .

  subsort Dog < Animal .
  subsorts Terrier Hound Toy Sporting < Dog .

  ops       bloodhound
      collie
      dalmatian
      pitbull
      schnauzer : -> Dog .

  ops penguin
      frog : -> Animal .

  op bromeliad : -> Vegetable .

  op breed : Dog Dog -> Dog [ctor] .

endfm



*********************************************************************
***  Chapter 2 - Functional Modules
***    Section 1 : Equations
***
***  PEANO-NAT-EXTRA : illustrates the fundamentals of equation
***  writing with the addition operator defined with Peano notation.
***
*********************************************************************

fmod PEANO-NAT-EXTRA is

  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor iter] .
  op _+_ : Nat Nat -> Nat .

  vars M N : Nat .

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

endfm
```

```
*************************************************************
***  Chapter 2 - Functional Modules
***    Section 1 : Equations
***
***  PEANO-NAT-MULT : multiplication operator for the Peano
***  natural numbers, a good illustration of recursive
***  algorithms in Maude.
***
*************************************************************

fmod PEANO-NAT-MULT is

  protecting PEANO-NAT-EXTRA .

  op _*_ : Nat Nat -> Nat .

  vars M N : Nat .

  eq N * 0 = 0 .
  eq N * s(M) = N + (N * M) .

endfm


set trace select on .

trace select (_*_) .

set trace on .

red s^3(0) * s^5(0) .

set trace off .



*************************************************************
***  Chap 2 - Functional Modules
***    Section 1 : Equations
***
***  LIST-SIZE : another example of recursive equation-writing
***  for lists.
***
*************************************************************

fmod LIST is

  sorts Elt List .
  subsort Elt < List .

  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .

endfm
```

```
fmod LIST-SIZE is

  protecting LIST .
  protecting PEANO-NAT .

  op size : List -> Nat .

  var E : Elt .         var L : List .

  eq size(nil) = 0 .
  eq size(E L) = s(size(L)) .

endfm



*************************************************************************
***  Chapter 2 - Functional Modules
***    Section 1 : Equations
***
***  CARD-DECK : your basic deck-o'-cards module, illustrating the
***   sort of "access" function that picks apart a constructor, with
***   CardNum and CardSuit.
***
*************************************************************************



fmod CARD-DECK is

   sorts Number Suit Card .

   ops A 2 3 4 5 6 7 8 9 10 J Q K : -> Number [ctor] .
   ops Clubs Diamonds Hearts Spades : -> Suit [ctor] .
   op _of_ : Number Suit -> Card [ctor] .

   op CardNum : Card -> Number .
   op CardSuit : Card -> Suit .

   var N : Number .   var S : Suit .

   eq CardNum( N of S ) = N .
   eq CardSuit( N of S ) = S .

endfm
```

```
*************************************************************************
***  Chapter 2 - Functional Modules
***   Section 2 : Conditional Equations
***
***  IRRITABLE-PROFESSOR : illustrates the two uses of pattern
***   matching - as a search method or as a value assignment.
***
*************************************************************************


fmod IRRITABLE-PROFESSOR is

   protecting STRING .

   sorts Question Exclamation Interruption .
   subsorts Question Exclamation < Interruption .

   op _? : String -> Question [ctor] .
   op _! : String -> Exclamation [ctor] .

   op reply : Interruption -> String .

   var I : Interruption .

   ceq reply(I) = "Questions after class, please" if (S:String) ? := I .
***eq reply( S:String ?) = "Questions after class, please" .
   ceq reply( S:String !) = T:String if T:String := "Please be quiet!" .

endfm



*****************************************************************
***  Chapter 2 - Functional Modules
***   Section 4 : Membership Axioms
***
***  CRAZY-EIGHTS : illustrates the use of conditional membership
***   axioms using our card-deck example.
***
***  SUICIDE-KING : illustrates the use of membership axioms using
***   our card-deck example.
***
*****************************************************************


fmod CRAZY-EIGHTS is

  protecting CARD-DECK .

  sort WildCard .
  subsort WildCard < Card .

  var C : Card .

  cmb C : WildCard if CardNum(C) == 8 .

endfm
```

```
fmod SUICIDE-KING is

  protecting CARD-DECK .

  sort SuicideKing .
  subsort SuicideKing < Card .

  mb K of Hearts : SuicideKing .

endfm



*****************************************************************
***  Chapter 2 - Functional Modules
***   Section 4 : Membership Axioms
***
***  CARD-PAIR : uses the matching condition in a conditional
***   membership axiom.
***
*****************************************************************


fmod CARD-PAIR is

   protecting CARD-DECK .

   sorts Pair PokerPair .
   subsort PokerPair < Pair .

   op <_;_> : Card Card -> Pair [comm] .

   var N : Number .   var P : Pair .

   cmb P : PokerPair if < N of S:Suit ; N of S':Suit > := P .

endfm
```

```
*****************************************************************************
***  Chapter 2 - Functional Modules
***   Section 5 : Operator and Statement Attributes
***
***  TENNIS : illustrates the use of prec and gather flags with the
***   non-associative versus operator.
***
*****************************************************************************


fmod TENNIS is

   sort Player .

   op _vs_ : Player Player -> Player [comm prec 33 gather (e E)] .
   op _murders_ : Player Player -> Player [prec 31] .
   op _teases_ : Player Player -> Player [prec 34] .

   ops sampras agassi roddick : -> Player [ctor] .

   vars P1 P2 : Player .

   eq sampras vs agassi = sampras .
   eq sampras vs roddick = roddick .
   eq agassi vs roddick = agassi .

   eq P1 murders P2 = P1 .
   eq P1 teases P2 = P2 murders P1 .

endfm



*****************************************************************************
***  Chapter 2 - Functional Modules
***   Section 6 : Expanding on Chapter 1
***
***  DOG-RACING : expands on the ANIMALS example from Chapter 1 to
***   illustrate some of the usefulness of kinds.
***
*****************************************************************************


fmod DOG-RACING is

   protecting ANIMALS .
   protecting NAT .

   op race : Dog Dog -> Dog .
   op speed : Dog -> Nat .

   op jackRusselTerrier : -> Terrier .

   vars N M : [Dog] .

   eq speed(bloodhound) = 20 .
   eq speed(collie) = 25 .
```

```
   eq speed(dalmatian) = 30 .
   eq speed(pitbull) = 15 .
   eq speed(schnauzer) = 30 .
   eq speed(jackRusselTerrier) = 10 .

   eq speed(frog) = 5 .
   eq speed(penguin) = 5 .

   eq speed(breed(N, M)) = (speed(N) + speed(M)) quo 2 .

   ceq race(N, M) = N if speed(N) > speed(M) .
   eq race(N, M) = M [owise] .

endfm



*****************************************************************
***  Chapter 3 - System Modules
***   Section 1 : Rewrite Laws
***
***  CLIMATE : illustrates the fundamentals of a rewrite law.
***
*****************************************************************


mod CLIMATE is

  sort weathercondition .

  op sunnyday : -> weathercondition .
  op rainyday : -> weathercondition .

  rl [raincloud] : sunnyday => rainyday .

endm



*********************************************************************
***  Chapter 3 - System Modules
***   Section 1 : Rewrite Laws
***
***  CIGARETTES - this example of rewrite laws will be a running
***    example throughout the rest of the chapter and tutorial.
***
*********************************************************************


mod CIGARETTES is

   sort State .

   ops c b : -> State [ctor] .   *** cigarettes and butts
   op __ : State State -> State [ctor assoc comm] .

   rl [smoke] : c => b .
```

```
   rl [makenew] : b b b b => c .

endm



*****************************************************************
***  Chapter 3 - System Modules
***   Section 1 : Rewrite Laws
***
***  COUNTING-CIGARETTES : an expansion on the cigarettes example
***
*****************************************************************


mod COUNTING-CIGARETTES is

   protecting NAT .

   sort State .

   ops c b : Nat -> State [ctor] .   ***cigarettes and butts
   op __ : State State -> State [ctor assoc comm] .

   vars W X Y Z : Nat .

   rl [smoke] : c(X) => b(X + 1) .
   rl [makenew] : b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .

endm



**************************************************************************
**
***  Chapter 3 - System Modules
***   Section 1 : Rewrite Laws
***
***  ARCADE-CRANE : our other running example of a rewrite system, but this
one
***   doesn't terminate.
***
**************************************************************************
**


mod ARCADE-CRANE is

   protecting QID .

   sorts ToyID State .
   subsort Qid < ToyID .

   op floor : ToyID -> State [ctor] .
   op on : ToyID ToyID -> State [ctor] .
   op clear : ToyID -> State [ctor] .
   op hold : ToyID -> State [ctor] .
```

```
   op empty : -> State [ctor] .

   op 1 : -> State [ctor] .

   op _&_ : State State -> State [ctor assoc comm id: 1] .

   vars X Y : ToyID .

   rl [pickup] : empty & clear(X) & floor(X) => hold(X) .
   rl [putdown] : hold(X) => empty & clear(X) & floor(X) .
   rl [unstack] : empty & clear(X) & on(X, Y) => hold(X) & clear(Y) .
   rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X, Y) .

endm



*****************************************************************
***  Chapter 3 - System Modules
***   Section 2 : Conditional Rewrite Laws
***
***  CONDITIONAL-RULES : illustrates the several different types of
***   conditions that work for a rewrite rule.
***
*****************************************************************


mod CONDITIONAL-RULES is

   protecting NAT .

   sort State .

   ops a b c : Nat -> State [ctor] .
   op __ : State State -> State [ctor] .

   op _-_ : Nat Nat -> Nat .

   vars X Y : Nat .

   eq X - Y = if X > Y then sd(X, Y) else 0 fi .

***   crl [equation1] : a(X) => b(X - 1) if X > 0 .
***   crl [equation2] : a(X) => b(X - 1) if X > 0 == true .
***   crl [equation3] : a(X) => b(X - 1) if X > 0 = true .
***   crl [membership1] : a(X) => b(X - 1) if X :: NzNat .
***   crl [membership2] : a(X) => b(X - 1) if X : NzNat .
   crl [patternmatch] : a(X) => b(X - 1) if s(N:Nat) := X .
   crl [rewrite] : b(X) => c(X * 2) if a(X) => b(Y) .

endm
```

```
*********************************************************************
***  Chapter 4 - Full Maude & Object-Oriented Modules
***   Section 1 : Full Maude
***
***  SMILE-LOOP : illustrates the use of LOOP-MODE in a simple
***   input/output example
***
*********************************************************************


mod SMILE-LOOP is

   inc LOOP-MODE .

   subsort QidList < State .

   op startsmiling : -> System .
   op none : -> State .

   eq startsmiling = [nil, none, nil] .

   var S : State .      vars I O : QidList .

   rl [smile] : ['smile I, S, O] => [I, S, O ' ': '- '`)] .
   rl [wink] : ['wink I, S, O] => [I, S, O ' '; '- '`)] .
   rl [gawp] : ['gawp I, S, O] => [I, S, O ' '8 '- '0] .
   rl [elvis] : ['elvis I, S, O] => [I, S, O ' '? 'B '- '`)] .

endm



*********************************************************************
***  Chapter 4 - Full Maude & Object-Oriented Modules
***   Section 1 : Full Maude
***
***  MOOD-LOOP : a more complex version of the smile loop, which
***   makes use of the State argument of the i/o constructor.
***
*********************************************************************


mod MOOD-LOOP is

   inc LOOP-MODE .

   sort Mood .

   op <_;_> : Mood Mood -> State [ctor] .
   op init : -> System .

   ops happy shocked playful : -> Mood .
   op # : -> Mood .

   eq init = [nil, < # ; # >, nil] .

   vars I O : QidList .       var S : State .
```

```
   rl [in-smile] : ['smile I, S, O] => [I, < happy ; # >, O] .
   rl [in-gape] : ['gape I, S, O] => [I, < shocked ; # >, O] .
   rl [in-wink] : ['wink I, S, O] => [I, < playful ; # >, O] .

   rl [out-smile] : [I, < # ; happy >, O] => [I, < # ; # >, O ' ': '- '`)] .
   rl [out-gape] : [I, < # ; shocked >, O] => [I, < # ; # >, O ' '8 '- '0] .
   rl [out-wink] : [I, < # ; playful >, O] => [I, < # ; # >, O ' '; '- '`)] .

   rl [wow!] : < happy ; # > => < # ; shocked > .
   rl [hehe] : < happy ; # > => < # ; playful > .
   rl [hey!] : < playful ; # > => < # ; shocked > .

endm



*****************************************************************************
***  Chapter 4 - Full Maude & Object-Oriented Modules
***   Section 5 : Rewrite Laws in Object-Oriented Modules
***
***  RESTAURANT : illustrates the basics of messages and rewriting
***
*****************************************************************************


(omod RESTAURANT is

   protecting NAT .

   class TABLE | occupied : Bool , chairs : Nat .
   class OutDoorTABLE | next2heater : Bool .
   subclass OutDoorTABLE < TABLE .

   ops One Two Three : -> Oid .

   msgs sit@ heaterswitch : Oid -> Msg .
   msgs _borrowchairfrom_ changetables : Oid Oid -> Msg .

   vars A B : Oid .  vars M N : Nat .

   rl [sit] : sit@(A) < A : TABLE | occupied : false >
            => < A : TABLE | occupied : true > .

   rl [switchoff] : heaterswitch(A) < A : OutDoorTABLE | next2heater : true >
            => < A : OutDoorTABLE | next2heater : false > .

   rl [switchon] : heaterswitch(A) < A : OutDoorTABLE | next2heater : false >
            => < A : OutDoorTABLE | next2heater : true > .

   rl [move] : changetables(A, B) < A : TABLE | occupied : true >
                              < B : TABLE | occupied : false >
            =>    < A : TABLE | occupied : false >
                  < B : TABLE | occupied : true > .

   rl [takechair] : (A borrowchairfrom B) < A : TABLE | chairs : M >
                                  < B : TABLE | chairs : s N >
```

```
            =>    < A : TABLE | chairs : s M >
                  < B : TABLE | chairs : N > .

endom)


*************************************************************************
***  Chapter 4 - Full-Maude & Object-Oriented Modules
***   Section 5 - Rewrite Laws in Object-Oriented Modules
***
***  TILE-PUZZLE : illustrates some of the power of object-oriented
***   programming
***
*************************************************************************


(omod TILE-PUZZLE is

   sorts Num Value Coord .
   subsort Num < Value .

   ops One Two Three Four Five Six Seven Eight : -> Num .
   op empty : -> Value .

   ops 0 1 2 : -> Coord .

   ops s_ p_ : Coord -> Coord .

   op `(_`,_`) : Coord Coord -> Oid [ctor] .

   eq s 0 = 1 .
   eq s 1 = 2 .
   eq p 1 = 0 .
   eq p 2 = 1 .

   class TILE | val : Value .

   msg move : Oid Oid -> Msg .

   vars R1 R2 C1 C2 : Coord . var V : Num .

   crl [goleft] : move((R1, C1), (R1, C2))
                        < (R1, C1) : TILE | val : V >
                        < (R1, C2) : TILE | val : empty >
                  =>    < (R1, C2) : TILE | val : V >
                        < (R1, C1) : TILE | val : empty >
                  if C2 == p C1 .

   crl [goright] : move((R1, C1), (R1, C2))
                         < (R1, C1) : TILE | val : V >
                         < (R1, C2) : TILE | val : empty >
                  =>     < (R1, C1) : TILE | val : empty >
                         < (R1, C2) : TILE | val : V >
                  if C2 == s C1 .

   crl [goup] : move((R1, C1), (R2, C1))
```

```
                        < (R1, C1) : TILE | val : V >
                        < (R2, C1) : TILE | val : empty >
                  =>    < (R2, C1) : TILE | val : V >
                        < (R1, C1) : TILE | val : empty >
                  if R2 == p R1 .

   crl [godown] : move((R1, C1), (R2, C1))
                        < (R1, C1) : TILE | val : V >
                        < (R2, C1) : TILE | val : empty >
                  =>    < (R2, C1) : TILE | val : V >
                        < (R1, C1) : TILE | val : empty >
                  if R2 == s C1 .

endom)



*********************************************************
***  Chapter 5 - Parameterization
***
***  GRAPH etc. : the complete parameterization system
***   for the Graph example.  This is the example in the
***   tutorial, and also a more complex example involving
***   a search function.
***
*********************************************************

(fth GRAPH is

    sorts Node Edge .

    ops node1 node2 : Edge -> Node .

endfth)

(fmod NETWORK-SIMPLE(X :: GRAPH) is

   sorts NodeSet(X) EdgeSet(X) Network .
   subsort X@Node < NodeSet(X) .
   subsort X@Edge < EdgeSet(X) .

   op _net_ : NodeSet(X) EdgeSet(X) -> Network [ctor] .
   op __ : NodeSet(X) NodeSet(X) -> NodeSet(X) [ctor comm assoc id: nnil] .
   op __ : EdgeSet(X) EdgeSet(X) -> EdgeSet(X) [ctor comm assoc id: nil] .
   op nnil : -> NodeSet(X) [ctor] .
   op nil : -> EdgeSet(X) [ctor] .

   op _in_ : X@Node NodeSet(X) -> Bool .
   op _in_ : X@Edge EdgeSet(X) -> Bool .

   op search : Network X@Node X@Node -> X@Edge .

   vars N N' : X@Node .       var NS : NodeSet(X) .
   vars E E' : X@Edge .       var ES : EdgeSet(X) .

   eq N in (N' NS) = N == N' or N in NS .
   eq N in nnil = false .
```

```
   eq E in (E' ES) = E == E' or E in ES .
   eq E in nil = false .

endfm)


(fmod NETWORK-SEARCH(X :: GRAPH) is

   sorts NodeSet(X) EdgeSet(X) Pathway(X) PathSet Network .
   subsort X@Node < NodeSet(X) .
   subsort X@Edge < EdgeSet(X) .
   subsorts X@Edge < Pathway(X) < PathSet .

   op _net_ : NodeSet(X) EdgeSet(X) -> Network [ctor] .
   op __ : NodeSet(X) NodeSet(X) -> NodeSet(X) [ctor assoc comm id: nnil] .
   op __ : EdgeSet(X) EdgeSet(X) -> EdgeSet(X) [ctor assoc comm id: nil] .
   op _:::_ : PathSet PathSet -> PathSet [ctor assoc comm id: null prec 21] .
   op _;_ : Pathway(X) Pathway(X) -> Pathway(X) [ctor assoc id: none prec 41]
.
   op nnil : -> NodeSet(X) [ctor] .
   op nil : -> EdgeSet(X) [ctor] .
   op none : -> Pathway(X) [ctor] .
   op null : -> X@Edge [ctor] .

   op _in_ : X@Node NodeSet(X) -> Bool .

   op search : Network X@Node X@Node -> Pathway(X) .

   var NS : NodeSet(X) .      vars A B : X@Node .
   var ES : EdgeSet(X) .      var E : X@Edge .  var P : Pathway(X) .

   eq P ::: none = P .

   eq P ; E ; null = P ; null .
   eq null ; E ; P = null ; P .

   eq A in B NS = A == B or A in NS .
   eq A in nnil = false .

   eq search(NS net E ES, A, B) = if A in NS and B in NS then
      if node2(E) == B then   if node1(E) == A then E else
                        search(NS net ES, A, node1(E)) ; E
                        fi
                  else
          search(NS net ES, A, B) :::
            if node1(E) == A
            then E ; search(NS net ES, node2(E), B)
            else none
            fi
      fi
                                          else none fi .

   ceq search(NS net nil, A, B) = null if A =/= B .

endfm)
```

```
(fmod GREEK-GRAPH is

   protecting NAT .

   sort Letter .

   ops alpha beta gamma delta epsilon lambda theta phi : -> Letter .

   op source : Nat -> Letter .
   op target : Nat -> Letter .

   eq source(1) = epsilon .   eq target(1) = lambda .
   eq source(2) = beta .      eq target(2) = alpha .
   eq source(3) = alpha .     eq target(3) = gamma .
   eq source(4) = alpha .     eq target(4) = delta .
   eq source(5) = delta .     eq target(5) = epsilon .
   eq source(6) = theta .     eq target(6) = phi .
   eq source(7) = phi .       eq target(7) = theta .

endfm)


(view Graph-Greek from GRAPH to GREEK-GRAPH is

   sort Node to Letter .
   sort Edge to Nat .

   op node1 to source .
   op node2 to target .

endv)



*****************************************************************************
***  Chapter 5 - Parameterization
***
***  PAIRS and STACK : an example of multiple parameters, parameterized
***   views, and parameterized theories, using our Pair sort and our
***   Deck of Cards examples.
***
*****************************************************************************

(fth OELT is

   sort Elt .

   op _>=_ : Elt Elt -> Bool .

endfth)



(fmod OPAIRLIST(X :: OELT | Y :: TRIV) is

   sorts Pair(X | Y) OPairList .
   subsort Pair(X | Y) < OPairList .
```

```
   op _~_ : X@Elt Y@Elt -> Pair(X | Y) [ctor] .
   op _;_ : OPairList OPairList -> OPairList [ctor assoc id: nil] .
   op nil : -> OPairList [ctor] .

   op top : OPairList -> Pair(X | Y) .
   op add : Pair(X | Y) OPairList -> OPairList .

   vars A1 A2 : Y@Elt .  vars E1 E2 : X@Elt .  var O : OPairList .

   eq top((E1 ~ A1) ; O) = E1 ~ A1 .

   eq add((E1 ~ A1), (E2 ~ A2) ; O) =     if E1 >= E2
                                 then (E2 ~ A2) ; add((E1 ~ A1) , O)
                                 else (E1 ~ A1) ; (E2 ~ A2) ; O
                                 fi .
   eq add((E1 ~ A1), nil) = E1 ~ A1 .

endfm)


(view Triv-String from TRIV to STRING is

   sort Elt to String .

endv)

(view OElt-Nat from OELT to NAT is

   sort Elt to Nat .
   op _>=_ to _>=_ .

endv)



(fmod DECK-COLOR is

   sort Color .         ops red blue : -> Color .
   op _>=_ : Color Color -> Bool .

   eq red >= blue = true .
   eq blue >= red = false .
   eq red >= red = true .
   eq blue >= blue = true .

endfm)


(view OElt-Color from OELT to DECK-COLOR is

   sort Elt to Color .
   op _>=_ to _>=_ .

endv)
```

```
(fmod STACK(X :: TRIV) is

   sort Stack(X) .
   subsort X@Elt < Stack(X) .

   op _;_ : Stack(X) Stack(X) -> Stack(X) [ctor assoc id: end] .
   op end : -> Stack(X) [ctor] .

   op top : Stack(X) -> X@Elt .
   op pop : Stack(X) -> Stack(X) .
   op push : X@Elt Stack(X) -> Stack(X) .

   var E : X@Elt .      var S : Stack(X) .

   eq top(E ; S) = E .
   eq pop(E ; S) = S .
   eq push(E, S) = E ; S .

endfm)


(view Triv-Stack(X :: TRIV) from TRIV to STACK(X) is

    sort Elt to Stack(X) .    endv)



*****************************************************************************
***  Chapter 5 - Parameterization
***   Section 5 : Parameterized Views and Theories
***
***  FMAP : another example of parameterized views and theories
***
*****************************************************************************



(fth FUNCTION(X :: TRIV | Y :: TRIV) is

   op f : X@Elt -> Y@Elt .

endfth)

(fmod FMAP(F :: FUNCTION(X :: TRIV | Y :: TRIV)) is

   sorts IndSet(X) DepSet(Y) .
   subsort X@Elt < IndSet(X) .
   subsort Y@Elt < DepSet(Y) .

   op __ : IndSet(X) IndSet(X) -> IndSet(X) [ctor assoc comm id: ind-null] .
   op _;_ : DepSet(Y) DepSet(Y) -> DepSet(Y) [ctor assoc comm id: dep-null] .
   op ind-null : -> IndSet(X) [ctor] .
   op dep-null : -> DepSet(Y) [ctor] .

   op fmap : IndSet(X) -> DepSet(Y) .

   var N : X@Elt .      var NS : IndSet(X) .
```

```
   eq fmap(N NS) = f(N) ; fmap(NS) .
   eq fmap(ind-null) = dep-null .

endfm)

(fmod NEGATIVE is

   pr NAT . pr INT .

   op _timesneg1 : Nat -> Int .

   var N : Nat .

   eq N timesneg1 = - N .

endfm)


(view Triv-Nat from TRIV to NAT is sort Elt to Nat .  endv)

(view Triv-Int from TRIV to INT is sort Elt to Int .  endv)

(view Fun-Neg(X :: TRIV | Y :: TRIV) from FUNCTION(X | Y) to NEGATIVE is

   op f to _timesneg1 .

endv)


(red in FMAP(Fun-Neg(Triv-Nat | Triv-Int)) : fmap(5 9 11 2 0) .)



*****************************************************************
***  Chapter 6 - Metaprogramming
***   Section 6 : Internal Strategies
***
***  REWRITE-EXCEPT : a quick and easy internal strategy which makes use
***   of the META-LEVEL descent functions.
***
*****************************************************************


fmod REWRITE-EXCEPT is

  pr META-LEVEL .

  op name : Rule -> Qid .
  op rmv : RuleSet Qid -> RuleSet .
  op rewriteExcept : Module Term Qid Nat -> ResultPair .

  vars L Q : Qid .      var I : ImportList .
  var S : SortSet .     var SS : SubsortDeclSet .
  var O : OpDeclSet .   var M : MembAxSet .     var E : EquationSet .
  var RS : RuleSet .    var R : Rule .          vars T1 T2 : Term .
  var N : Nat .
```

```
  eq name(rl T1 => T2 [label(Q) A:AttrSet] .) = Q .
  eq name(crl T1 => T2 if C:Condition [label(Q) A:AttrSet] .) = Q .

  eq rmv(R RS, L) = if name(R) == L then RS else R rmv(RS, L) fi .
  eq rmv(none, L) = none .

  eq rewriteExcept(mod Q is I sorts S . SS O M E RS endm, T1, L, N) =
      metaRewrite(mod Q is I sorts S . SS O M E rmv(RS, L) endm, T1, N) .

endfm



*****************************************************************
***  Chapter 6 - Metaprogramming
***   Section 6 - Internal Strategies
***
***  SANS : an elaboration of the rewriteExcept strategy
***
*****************************************************************


fmod SANS is

   pr META-LEVEL .

   op sans : Module QidList -> Module .
   op rmvM : MembAxSet Qid -> MembAxSet .
   op rmvE : EquationSet Qid -> EquationSet .
   op rmvR : RuleSet Qid -> RuleSet .

   vars N Q : Qid .     var QL : QidList .
   var I : ImportList .
   var S : SortSet .    var SS : SubsortDeclSet .
   var O : OpDeclSet .  var T1 T2 : Term .
   var M : MembAx .     var MS : MembAxSet .    var X : Sort .
   var E : Equation .   var ES : EquationSet .  var A : AttrSet .
   var R : Rule .       var RS : RuleSet .      var C : Condition .

   eq sans(mod N is I sorts S .  SS O MS ES RS endm, Q QL) =
      sans(mod N is I sorts S . SS O rmvM(MS, Q)
                                   rmvE(ES, Q)
                                   rmvR(RS, Q) endm, QL) .

   eq rmvM(mb T1 : X [label(Q) A] . MS, Q) = MS .
   eq rmvM(cmb T1 : X if C [label(Q) A] . MS, Q) = MS .
   eq rmvM(M MS, Q) = M rmvM(MS, Q) [owise] .
   eq rmvM(none, Q) = none .

   eq rmvE(eq T1 = T2 [label(Q) A] . ES, Q) = ES .
   eq rmvE(ceq T1 = T2 if C [label(Q) A] . ES, Q) = ES .
   eq rmvE(E ES, Q) = E rmvE(ES, Q) [owise] .
   eq rmvE(none, Q) = none .

   eq rmvR(rl T1 => T2 [label(Q) A] . RS, Q) = RS .
   eq rmvR(crl T1 => T2 if C [label(Q) A] . RS, Q) = RS .
```

```
   eq rmvR(R RS, Q) = R rmvR(RS, Q) [owise] .
   eq rmvR(none, Q) = none .

endfm



*******************************************************************************
***  Chapter 6 - Metaprogramming
***   Section 6 : Internal Strategies
***
***  CIGARETTE-TEST : the Grand Finale of Maude examples, implementing
***   an internal strategy in an optimization problem based on our
***   Chapter 3 cigarettes example.
***
*******************************************************************************


mod CIGARETTES-3 is

   pr NAT .

   sorts Brand State .


   ops DROMEDARY OLDPORT SCARBOROUGH-MAN MISSOURI-FATS : -> Brand .
   ops c b : Brand Nat -> State [ctor] .
   op 1 : -> State [ctor] .
   op fail : -> Brand .
   op __ : State State -> State [ctor assoc comm id: 1] .

   op price : Brand -> Nat .

   vars V X W Y Z : Nat .
   var B : Brand .

   eq price(DROMEDARY) = 312 .
   eq price(OLDPORT) = 312 .
   eq price(SCARBOROUGH-MAN) = 364 .
   eq price(MISSOURI-FATS) = 418 .

   eq price(fail) = 0 .

   rl [smoke] : c(B, X) => b(B, X + 1) .

   rl [makenew] : b(DROMEDARY, V)
              b(DROMEDARY, W)
              b(DROMEDARY, X)
              b(DROMEDARY, Y)
              b(DROMEDARY, Z) => c(DROMEDARY, V + W + X + Y + Z) .
   rl [makenew] : b(OLDPORT, W)
              b(OLDPORT, X)
              b(OLDPORT, Y)
              b(OLDPORT, Z) => c(OLDPORT, W + X + Y + Z) .
   rl [makenew] : b(SCARBOROUGH-MAN, X)
              b(SCARBOROUGH-MAN, Y)
              b(SCARBOROUGH-MAN, Z) => c(SCARBOROUGH-MAN, X + Y + Z) .
```

```
   rl [makenew] : b(MISSOURI-FATS, Y)
              b(MISSOURI-FATS, Z) => c(MISSOURI-FATS, Y + Z) .

endm


mod CIGARETTE-VENDING-MACHINE is

   protecting CIGARETTES-3 .

   sort BrandSet .
   subsort Brand < BrandSet .

   op vendor : BrandSet -> State .
   op _&_ : BrandSet BrandSet -> BrandSet [ctor assoc comm id: null] .
   op null : -> BrandSet [ctor] .

   var B : Brand .
   var BS : BrandSet .

   rl [buypack] : vendor(B & BS) =>
      c(B, 0)     c(B, 0)     c(B, 0)     c(B, 0)     c(B, 0)
      c(B, 0) c(B, 0)   c(B, 0)     c(B, 0)     c(B, 0)
      c(B, 0) c(B, 0)   c(B, 0) c(B, 0) c(B, 0)
      c(B, 0) c(B, 0) c(B, 0) c(B, 0) c(B, 0) .

endm


fmod CIGARETTE-TEST is

   pr META-LEVEL .
   pr CIGARETTE-VENDING-MACHINE .

   sorts CigTotal CigTotalSet .
   subsort CigTotal < CigTotalSet .

   op ct : Term Term -> CigTotal [ctor] .
   op __ : CigTotalSet CigTotalSet -> CigTotalSet [ctor assoc comm id: none]
.
   op none : -> CigTotalSet [ctor] .

   op goTest! : -> CigTotalSet .
   op cigApply : Nat -> ResultTriple .
   op cigTest : Nat -> CigTotalSet .

   op total : ResultTriple -> CigTotal .
   op getbrand : Substitution -> Term .
   op getnum : ResultPair -> Term .
   op extract : ResultPair -> Term .

   var TL : TermList .
   var T : Term . var Y : Type .          var SB : Substitution .
   vars Q1 Q2 : Term .  var M : Module .  var N : Nat .

   eq goTest! = cigTest(0) .
```

```
   eq cigApply(N) = metaApply(['CIGARETTE-VENDING-MACHINE],
      'vendor['_&_['DROMEDARY.Brand, '_&_['OLDPORT.Brand,
            '_&_['SCARBOROUGH-MAN.Brand, 'MISSOURI-FATS.Brand]]]],
                  'buypack, none, N) .

   eq cigTest(N) = if cigApply(N) == (failure).ResultTriple?
               then none
               else total(cigApply(N)) cigTest(N + 1)
               fi .

   eq total({T, Y, SB}) = ct( getbrand(SB),
                        extract( metaReduce( ['CIGARETTES-3],
            '_quo_['price[getbrand(SB)], getnum(
                        metaRewrite(['CIGARETTES-3], T, 1000)
                                    )
                  ]                 )
                           )
                        ) .

   eq getbrand( Q1 <- Q2 ; SB ) = if Q1 == 'B:Brand then Q2
                           else getbrand(SB) fi .
   eq getbrand( none ) = 'fail.Brand .

   eq getnum( { '__['b[Q1, Q2] , TL ] , Y } ) = '_+_[Q2, getnum({TL, Y})] .
   eq getnum( { '__['b[Q1, Q2] , T] , Y } ) = '_+_[Q2, getnum({T, Y})] .
   eq getnum( { ('b[Q1, Q2] , TL) , Y } ) = '_+_[Q2, getnum({TL, Y})] .
   eq getnum( { 'b[Q1, Q2] , Y } ) = Q2 .

   eq extract( { T , Y } ) = T .

   eq ct( 'fail.Brand, T ) = none .

endfm
```