9. Atomic Semantics of Spec

This document defines the semantics of the atomic part of the Spec language fairly carefully. It tries to be precise about all difficult points, but is sloppy about some things that seem obvious in order to keep the description short and readable. For the syntax and an informal account of the semantics, see the Spec reference manual, handout 4.

There are three reasons for giving a careful semantics of Spec:

1. To give a clear and unambiguous meaning for Spec programs.
2. To make it clear that there is no magic in Spec; its meaning can be given fairly easily and without any exotic methods.
3. To show the versatility of Spec by using it to define itself, which is quite different from the way we use it in the rest of the course.

This handout is divided into two parts. In the first half we describe semi-formally the essential ideas and most of the important details. Then in the second half we present the atomic semantics precisely, with a small amount of accompanying explanation.

Semi-formal atomic semantics of Spec[1]

Our purpose is to make it clear that there is no arm waving in the Spec notation that we have given you. A translation of this into fancy words is that we are going to study a formal semantics of the Spec language.

Now that is a formidable sounding term, and if you take a course on the semantics of programming languages (6.821—Gifford, 6.830J—Meyer) you will learn all kinds of fancy stuff about bottom and stack domains and fixed points and things like that. You are not going to see any of that here. We are going to do a very simple minded, garden-variety semantics. We are just going to explain, very carefully and clearly, how it is that every Spec construct can be understood, as a transition of a state machine. So if you understand state machines you should be able to understand all this without any trouble.

One reason for doing this is to make sure that we really do know what we are talking about. In general, descriptions of programming languages are not in that state of grace. If you read the Pascal manual or the C manual carefully you will come away with a number of questions about exactly what happens if I do this and this, questions which the manual will not answer adequately. Two reasonably intelligent people who have studied it carefully can come to different conclusions, argue for a long time, and not be able to decide what is the right answer by reading the manual.

[1] These semi-formal notes take the form of a dialogue between the lecturer and the class. They were originally written by Mitchell Charity for the 1992 edition of this course, and have been edited for this handout.

There is one class of mechanisms for saying what the computer should do that often does answer your questions precisely, and that is the instruction sets of computers (or, in more modern language, the architecture). These specs are usually written as garden variety state machines with fairly simple transitions, which is not beyond the power of the guy who is writing the manual to describe properly. A programming language, on the other hand, is not like that. It has much more power, generality, and wonderfulness, and also much more room for confusion.

Another reason for doing this is to show you that our methods can be applied to a different kind of system than the ones we usually study, that is, to a programming language, a notation for writing programs or a notation for writing specifications. We are going to learn how to write a spec for that particular class of computer systems. This is a very different application of Spec from the one we have just finished looking at, which was file systems. For describing a programming language, Spec is not the ideal descriptive notation. If you were in the business of giving the semantics of programming languages, you wouldn't use Spec. There are many other notations, some of them better than Spec (although most are far worse). But Spec is good enough; it will do the job. And there is a lot to be said for just having one notation you can use over and over again, as opposed to picking up a new one each time. There are many pitfalls involved in devising a new notation.

Those are the two themes of this lecture. We are going to get down to the foundations of Spec, and we are going to see another, very different application of Spec. Certainly a programming language is very different from a file system.

For this lecture, we will only talk about the sequential or atomic semantics of Spec, not about concurrent semantics. Consider the program:

```
                    x, y = 0
thread 1:                                   thread 2:
<<  x := 3 >>                               << z := x + y >>
<<  y := 4 >>
```

In the concurrent world, it is possible to get any of the answers 0, 3, or 7. In the sequential world, which we are in today, the only possible answers are 0 and 7. It is a simpler world. We will be talking later (in handout 17 on formal concurrency) about the semantics of concurrency, which is unavoidably more complicated.

In a sequential Spec program, there are three basic constructs (corresponding to sections 5, 6, and 7 of the reference manual):

Expressions
Commands
Routines

In order to describe what each of these things means, we first of all need some notion of what kind of thing the meaning of an expression or command might be. Then we have to explain in detail exactly what the meaning of each possible kind of expression is. The basic technique we use is the standard one for a situation where you have things that are made up out of smaller things: structural induction.

The idea of structural induction is this. If you have something which is made up of an A and a B, and you know the meaning of each, and have a way to put them together, you know how to get the meaning of the bigger thing.

Some ways to put things together in Spec:

```
A , B
A ; B
a + b
A [] B
```

State

What are the meanings going to be? Our basic notion is that what we are doing when writing a Spec program is describing a state machine. The central properties of a state machine are that it has states and it has transitions.

State is a function of names to values: State: Name -> Value. For example:

```
VAR  x: Int
     y: Int
```

If there are no other variables, the state simply consists of the mapping of the names "x" and "y" to their corresponding values. Initially, we don't know what their values are. Somehow the meaning we give to this whole construct has to express that.

Next, if we write `x := 1`, after that the value of x is 1. So the meaning of this had better look something like a transition that changes the state, so that no matter what the x was before, it is 1 afterwards. That's what we want this assignment to mean.

Spec is much simpler than C. In particular, it does not have "references" or "pointers". When you are doing problems, if you feel the urge to call `malloc`, the correct thing to do is to make a function whose range is whatever sort of thing you want to allocate, and then choose a new element of the domain that isn't being used already. You can use the integers or any convenient sort of name for the domain, that is, to name the values. If you define a CLASS, Spec will do this for you automatically.

So this is the state, just these name to value mappings.

Names

Spec has a module structure. The names can have two parts. When referring to a variable in another module, both parts must be used.

```
MODULE A                    MODULE B

VAR  x                       A.x := 3
     x := 3
     ...
     A.x := 3
```


From the point of view of the semantics, we are going to use A.x as the name everywhere. In order to apply the semantics, you first must go through the program and replace every x declared in the current module A with A.x. You convert all references to global variables to these two part names, so that each name refers to exactly one thing. This makes things simpler to describe and understand, but uglier to read. This transformation doesn't change the meaning of the program; it could have been written with two part names in the first place.

All the global variables have these two part names. However, local variables are not prefixed by the module name:

```
PROC
     VAR i | ... i
```

This is how we tell the global state apart from the local state. Global state has dots, local state does not.

Question: Can modules be nested?

No. Spec is meant to be suitable for the kinds of specs and implementations that we do in this course, which are no more than moderately complex. Features not really needed to write our specs are left out to keep it simpler.

Expressions

What should the meaning of an expression be? Note that expressions do *not* affect the state.

Question: What about assignments?

Assignments are not expressions. If you have been programming in C, you have the weird idea that assignments are expressions. But not in the rest of the world. Spec in particular takes a hard line that not only are assignments not expressions, but functions are not allowed to affect the state.

What are the semantics of an expression? Well, the type for the meaning of an expression is `S -> V`. An expression is a function from state to value. It can be a partial function, since Spec does not require that all expressions be defined. But it has to be a function—we do require that expressions are deterministic. We want determinism so something like `f(x) = f(x)` always comes out true. Reasoning is just too hard if this isn't true.. If a function is nondeterministic then obviously this needn't come out true. (The classic example of a nondeterministic function is a random number generator.)

So, **expressions are deterministic and do not affect state.**

There are three types of expressions:

Type	*Example*	*Meaning*
constant	`1`	`(\ s | 1)`
variable	`x`	`(\ s | s("x"))`
function invocation	`f(x)`	next sub-section

(The type of these lambda's is not quite right, as we will see later).

Note that we have to keep the Spec in which we are writing the semantics separate from the Spec of the semantics we are describing. Therefore, we had to write `s("x")` instead of just `x`, because it is the `x` of the target system we are talking about, not the `x` of the describing system.

The third type of expression is function invocation. We will only talk about functions with a single argument. If you want a function with two arguments, you can make one by combining the two arguments into a tuple or record, or by currying: defining a function of the first argument that returns a function of the second argument.

What about `x + y`? This is just shorthand for `T."+"(x, y)`, where `T` is the type of `x`. Everything that is not a constant or a variable is an invocation. This should be a familiar concept for those of you who know Scheme.

Semantics of function invocation

What are the semantics of function invocation? Given a function `T -> U`, the correct type of its meaning is `(T, S) -> U`, since the function can read the state but not modify it. Next, how are we going to attach a meaning to an invocation `f(x)`? Remember the rule of structural induction. In order to explain the meaning of a complicated thing, you are supposed to build it out of the meaning of simpler things. We know the meaning of `x` and of `f`. We need to come up with a map of states to values that is the meaning of `f(x)`. Somehow we are going to need to get our hands on the meaning of `f` and the meaning of `x`, and then to put them together appropriately. What is the meaning of `f`? `s("f")`. So,

`f(x)` means `... s("f") ... s("x")`

How are we going to put it together, remembering the type we want for `f(x)`?

`f(x)` means `(\ s | s("f") (s("x"), s)) )`

Now this could be complete nonsense, for instance if `s("f")` evaluates to an integer. If `s("f")` isn't a function then this doesn't typecheck. But there is no doubt about what this means if it is legal. It means invoke the function.

That takes care of expressions, because there are no other expressions besides these. Structural induction says you work your way through all the different ways to put little things together to make big things, and when you have done them all, you are finished.

Question: What about undefined functions?

Then the `(T, S) -> U` mapping is partial.

Question: Is `f(x) = f(x)` if `f(x)` is undefined?

No, it's undefined. But those are deep waters and I propose to stay out of them.

Commands

What is the type of the meaning of a command? Well, we have states and values to play with, and we have used up `S -> V` on expressions. What sort of thing is a command? It's a transition from one state to another.

Expressions `S -> V`

Commands `S -> S ?`

This is good for a subset of commands. But what about this one?

```
x := 1 [] x := 2
```

Is its meaning a function from states to states? No, from states to *sets* of states. It can't just be a function. It has to be a relation. Of course, there are lots of ways to code relations as functions. The way we use is:

Commands `(S, S) -> Bool`

There is a small complication because Spec has exceptions, which are useful for writing many kinds of specifications, not to mention programs. So we have to deal with the possibility that the result of a command is not a garden-variety state, but involves an exception.

To handle this we make a slight extension and invent a thing called an *outcome*, which is very much like a state except that it has some way of coding that an exception has happened. Again, there are many ways to code that. The way we use is that an outcome has the same type as a state: it's a function of names to values. However, there are a couple of funny names that you can't actually write in the program. One of them is `$x`, and we adopt the convention that if `o("$x") = ""` (empty string), then `o` is a garden-variety state. If `o("$x") = "exception-name"`, then there is that exception in outcome `o`. Some Spec commands, in particular `";"` and `EXCEPT`, do something different if one of their pieces produces an exception.

Now we just work our way through the command constructs (with an occasional digression).

Commands — assignment

```
x := 1
```

or in general

```
variable := expression
```

What we have to come up with for the meaning is an expression of the form

```
(\ s, o | RET ...)
```

How do we say that o is related to s? The function returns true. We are encoding a relation between states and outcomes as a function from a state and outcome to a Bool. The function is supposed to give back true exactly when the relation holds.

So when does the relation hold for x := exp? Well, perhaps when o(x) = exp? (ME is the meaning function for expressions.)

```
o("x") = ME(e)(s)
```

. Well, the valid transition

```
x=0                x=1
          ->
y=0                y=0
```

would certainly be allowed. But what others would be allowed? What about:

```
x=0                x=1
          ->
y=0                y=94
```

It would also be allowed, so this can't be quite right. Half right, but missing something important. You have to say that you don't mess around with the rest of the state. The way you do that is to say that the outcome is equal to the state except at the variable.

```
o = s{"x" -> ME(e)(s)}
```

This is just a Spec function constructor, of the form f{arg -> value}.

Aside—an alternate encoding for commands

As we said before, there are many ways to code the command relation. Another possibility is:

Commands S -> SET S

This encoding seems to make the meanings of commands clumsier to write, though it is entirely equivalent to the one we have chosen.

There is a third approach, which has a lot of advantages: write predicates on the state values. If x and y are the state variables in the prestate, and x' and y' the state variables in the poststate, then

```
(x' = 1 /\ y' = y)
```

is another way of writing

```
o = s{"x" -> 1}
```

In fact, this approach is another way of writing programs. I could write everything just as predicates. Of course, I could also write everything as this o = s{...} sort of cruft, but that would look pretty awful. It is more conceivable that we would want to write things as predicates, because it doesn't look so bad.

Sometimes it's actually nice to do this. Say you want to write the predicate that says you can have any value at all for x. The spec


```
VAR z | x := z
```

is just

```
(y' = y)
```

(in the simple world where the only state variables are x and y). This is much simpler that the previous, rather inscrutable, piece of program. So sometimes this predicate way of doing things can be a lot nicer, but in general it seems to be not as satisfactory, mainly because the y'=y stuff clutters things up an awful lot.

That was just an aside, but sometimes it's convenient to describe the things that can go on in a specification using predicates rather than functions from state pairs to Bool.

Commands — routine invocation p(x)

What are the semantics of routine invocation? Well, it has to do something with s. What about the argument? There are many ways to deal with the argument. The way we do it is to use another pseudo-variable $a to pass the argument and get back the result.

The meaning of p(e) is going to be

```
LAMBDA (s, o) = RET          (s                          Take the state,
                               {"$a" -> ME(e)(s)}        append the argument,
                    ME(p)(s)                             get the routine
                                                , o)     and invoke it
```

or, writing the whole thing on one line in the normal way,

```
LAMBDA (s, o) = RET ME(p)(s)(s{"$a" -> ME(e)(s)}, o)
```

What does this say? This invocation relates a state to an outcome if, when you take that state, and modify its $a component to be equal to the value of the argument, the meaning of the routine relates that state to the outcome. Another way of writing this, which isn't so nested and might be clearer, would be to introduce an intermediate state s':

```
VAR s' = s{"$a" -> ME(e)(s)} | ME(p)(s)(s',o)
```

These two are exactly the same thing. The invocation relates s to o iff the routine relates s' to o, where s'' is just s with the argument passing component modified. $a is just a way of communicating the argument value to the routine.

Question: Why use ME(p)(s) rather than MR?

MR is the meaning function for routines, that is, it turns the syntax of a routine declaration into a function on states and arguments that is the meaning of that syntax. We would use MR if we were looking at a FUNC. But p is just a variable (of course it had better be bound to a routine value, or this won't typecheck).

Aside—an alternate encoding for invocation

Here is a different way of communicating the argument value to the function. We could take the view that the routine definition

```
PROC P(i: Int) = ...
```

is defining a whole flock of different commands, one for every possible argument value. Then we need to pick out the right one based on the argument value we have. If we coded it this way (and it is merely a coding thing) we would get:

```
ME(p)(s)(ME(e)(s)) (s,o)
```

This says, first get `ME(p)`, the meaning of `p`. This is not a transition but a function from argument values to transitions, because the idea is that for every possible argument value, we are going to get a different meaning for the routine, namely what that routine does when given that particular argument value. So we pass it the argument value `ME(e)(s)`, and invoke the resulting transition.

These two alternatives are based on different choices about how to code the meaning of routines. If you code the meaning of a routine simply as a transition, then Spec picks up the argument value out of the magic `$a` variable. But there is nothing mystical going on here. Setting `$a` corresponds exactly to what we would do if we were designing a calling sequence. We would say "I am going to pass the argument in register 1". Here, register 1 is `$a`.

The second approach is a little bit more mystical. We are taking more advantage of the wonderful abstract power and generality that we have. If someone writes a factorial function, we will treat it as an infinite supply of different functions; one computes the factorial of 1, another the factorial of 2, another the factorial of 3, and so forth. In `ME(p)(s)(ME(e)(s))(s, o)`, `ME(p)(s)` is the infinite supply, `ME(e)(s)` is the argument that picks out a particular function, to which we finally pass `(s, o)`.

However, there are lots of other ways to do this. One of the things which makes the semantics game hard is that there are many choices you can make. They don't really make that much difference, but they can create a lot of confusion, both because a bad choice can leave you in a thicket of notation, and because you can get confused about what choice was made.

So, while this

```
RET ME(p) (S) (S("$a" -> ME(e) (s)),o)
```

and this

```
VAR s' := s{"$a" -> ME(e)(s)} | RET ME(p)(s) (s',o)
```

are two ways of writing exactly the same thing, this

```
ME(p)(s)(ME(e)(s)) (s,o)
```

is different, and only makes sense with a different choice about what the meaning of a function is. The latter is more elegant, but we use the former because it is less confusing.

Stepping back from these technical details, what the meaning function is doing is taking an expression and producing its meaning. The expression is a piece of syntax, and there are a lot of possible ways of coding the syntax. Which exact way we chose isn't that important.

Commands — `SKIP`

```
LAMBDA (s, o) = RET s = o
```

In other words, the outcome after `SKIP` is the same as the prestate. Later on, in the formal half of the handout, we give a table for the commands which takes advantage of the fact that there is a lot of boilerplate—the `LAMBDA (s, o) = RET` stuff is always the same, and so is the treatment of exceptions. So the table just shows, for each syntactic form, what goes after the `<M:RET:M>`.

Commands — *`HAVOC`*

```
LAMBDA (s, o) = true
```

In other words, after `HAVOC` you can have any outcome. Actually this isn't quite good enough, since we want to be able to have any *sequence* of outcomes. We deal with this by introducing another magic state component `$havoc` with a `Bool` value. Once `$havoc` is true, any transition can happen, including one that leaves it true and therefore allows havoc to continue. We express this by adding to the command boilerplate the disjunct `s("$havoc")`, so that if `$havoc` is true in `s`, any command relates `s` to any `o`.

Now for the compound commands.

Commands — `c1 [] c2`

```
MC(c1)          MC(c2)
     (s, o)          (s, o)
           \/
```

or on one line,

```
MC(c1)(s, o) \/ MC(c2)(s, o)
```

Non-deterministic choice is the 'or' of the relations.

Commands — `c1 [*] c2`

It is clear we should begin with

```
MC(c1)(s, o) \/ …
```

But what next? One possibility is

```
~ MC(c1)(s, o) ==> ...
```

This is in the right direction, but not correct. Else means that if there is no *possible* outcome of `c1`, then you get to try `c2`. So there are two possible ways for an else to relate a state to an outcome. One is for `c1` to relate the state to the outcome, the other is that there is no possible way to make progress with `c1` in the state, and `c2` to relates the state to the outcome.

The correct encoding is

```
MC(cl)(s,o) \/ (ALL o' | ~ MC(cl) (s, o')) /\ MC(c2)(s,o) )
```

Commands — `c1 ; c2`

Although the meaning of semicolon may seem intuitively obvious, it is more complex than one might first suspect—more complicated than or, for instance. We interpret the command `c1 [] c2` as `MC(cl) \/ MC(c2)`. Because semicolon is a sequential composition, it requires that our semantics move through an intermediate state.

If these were functions (if we could describe the commands as functions) then we could simply describe a sequential composition as `(F2 (Fl s))`. However, because Spec is not a functional language, we need to compose relations, in other words, to establish an intermediate state as a precursor to the final output state. As a first attempt, we might try:

```
(LAMBDA (s, o) -> Bool = RET
    (EXISTS o' | MC(cl)(s, o') /\ MC(c2)(o', o)))
```

In words, this says that you can get from `s` to `o` via `c1 ; c2` if there exists an intermediate state `o'` such that `c1` takes you from `s` to `o'` and `c2` takes you from `o'` to `o`. This is indeed the composition of the relations. But is this always the meaning of `";"`? In particular, what if `c1` produces an exception? When `c1` produces an exception, we should *not* execute `c2`. Our first try does not capture that possibility. To correct for this, we need to verify that `o'` is a normal state. If it is an exceptional state, then it is the result of the composition and we ignore `c2`.

```
    (EXISTS o' | MC(cl)(s, o') /\ (   ~IsX(o') /\ MC(c2)
                                  \/  IsX(o') /\ o' = o))
```

Commands — `c1 EXCEPT xs => c2`

Now, what if we have a handler for the exception? If we assume (for simplicity) that all exceptions are handled, we would simply implement the complement of the semicolon case. If we get an exception, then do `c2`. If there is no exception, do *not* do `c2`. We also need to include an additional check to insure that the exception considered is an element of the exception set—that is to say, that it is a handled exception.

```
    (EXISTS o' | MC(cl)(s, o') /\
              (   ((~IsX(o') \/ ~o'("$x") IN xs) /\ o' = o)
               \/    IsX(o') /\  o'("$x") IN xs) /\ MC(c2)(o'{"$x" -> ""}, o)
    )
```

So, with this semantics for handling exceptions, the meaning of:

```
 (c1 EXCEPT xs => c2); c3)
```

is

if normal	do `c1`, no `c2`, do `c3`
if exception, handled	do `c1`, do `c2`, do `c3`
if exception and not handled	do `c1`, no `c2`, no `c3`

Commands — `VAR id: T | c0`

The idea is “there exists a value for `id` such that `c0` succeeds”. This intuition suggests something like

```
    (EXISTS v |  v IN T /\ MC(c0)(s{"id" -> v}, o))
```

However, if we look carefully, we see that `id` is left defined in the output state `o`. (Why is this bad?) To correct this omission we need to introduce an intermediate state `o'` from which we may arrive at the final output state `o` where `id` is undefined.

```
    (EXISTS v, o' | v IN T /\ MC(c0)(s("id" -> v}, o') /\ o = o'(id -> })
```

Routines

In Spec, routines include functions, atomic procedures, and procedures. For simplicity, we focus on atomic procedures. How do we think about `APROC`s?

We know that the body of an `APROC` describes transitions from its input state to its output state. Given this transition, how do we handle the results? We introduce a pseudo name `$a` to which a procedure’s argument value is bound, and the caller also collects the value from `$a` after the procedure body’s transition. Refer to the definition of `MR` below for a more complete discussion.

In reality, Spec is more complex because it attempts to make `RET` more convenient by allowing it to occur anywhere in a routine. To accommodate this, the meaning of `RET e` is to set `$a` to the value of `e` and then raise the special exception `$RET`, which is handled as part of the invocation.

Formal atomic semantics of Spec

In the rest of the handout, we describe the meaning of atomic Spec commands in complete detail, except that we do not give precise meanings for the various expression forms other than lambda expressions; for the most part these are drawn from mathematics, and their meanings should be clear. We also omit the detailed semantics of modules, which is complicated and uninteresting.

Overview

The semantics of Spec are defined in three stages: expressions, atomic commands, and non-atomic commands (treated in handout 17 on formal concurrency). For the first two there is no concurrency: expressions and atomic commands are atomic. This makes it possible to give their meanings quite simply:

> Expressions as *functions* from states to results, that is, values or exceptions.
>
> Atomic commands as *relations* between states and outcomes: a command relates an initial state to every possible outcome of executing the command in the initial state.

An outcome maps names (treated as strings) to values. It also maps three special strings that are not program names (we call them pseudo-names):
`$a`, which is used to pass argument and result values in an invocation;
`$x`, which records an exceptional outcome;
`$havoc`, which is true if any sequence of later outcomes is possible.

A state is a normal outcome, that is, an outcome which is not exceptional; it has `$x=noX`. The looping outcome of a command is encoded as the exception `$loop`; since this is not an identifier, you can't write it in a handler.

The state is divided into a *global* state that maps variables of the form `m.id` (for which `id` is declared at the top level in module `m`) and a *local* state that maps variables of the form `id` (those whose scope is a `VAR` command or a routine). Routines share only the global state; the ones defined by `LAMBDA` also have an initial local state, while the ones declared in a `routineDecl` start with an empty local state. We leave as an exercise for the reader the explicit construction of the global state from the collection of modules that makes up the program.

We give the meaning of a Spec program using Spec itself, by defining functions `ME`, `MC`, and `MR` that return the meaning of an expression, command, and routine. However, we use only the functional part of Spec. Spec is not ideally suited for this job, but it is serviceable and by using it we avoid introducing a new notation. Also, it is instructive to see how the task is writing this particular kind of specification can be handled in Spec.

You might wonder how this specification is related to an implementation of Spec, that is, to a compiler or interpreter. It does look a lot like an interpreter. As with other specifications written in Spec, however, this one is not a practical implementation because it uses existential quantifiers and other forms of non-determinism too freely. Most of these quantifiers are just there for clarity and could be replaced by explicit computations of the needed values without much difficulty. Unfortunately, the quantifier in the definition of `VAR` does not have this property; it actually requires a search of all the values of the specified type. Since you have already seen that we don't know how to give a practical implementation of Spec, it shouldn't be surprising that this handout doesn't contain one.

Note that before applying these rules to a Spec program, you must apply the syntactic rewriting rules for constructs like `VAR id := e` and `CLASS` that are given in the reference manual. You must also replace all global names with their fully qualified forms, which include the defining module, or `Global` for names declared globally (see section 8 of the reference manual).

Terminology

We begin by giving the types and special values used to represent the Spec program whose meaning is being defined. We use two methods of functions, `+` (overlay) and `restrict`, that are defined in section 9 of the reference manual.


```
TYPE V          =  (Routine + ...)                      % Value
     Routine    =  aTr                                  % defined as the last type below

     Id         =  String                               % Identifer
                   SUCHTHAT (\ id | (EXISTS c: Char, s1: String, s2: String |
                         id = {c} + s1 + s2 /\ c IN letter + digit
                      /\ s1.set <= letter+digit+{'_'} /\ s2.set <= {''} ))
     Name       =  String
                   SUCHTHAT (\ name | name IN   ids + globals
                                              + {"$a", "$x", "$havoc"})
     X          =  String                               % eXception
                   SUCHTHAT (\ x | x IN ids + {noX, retX, loopX, typeX})
     XS         =  SET X                                % eXception Set

     O          =  Name -> V WITH {isX:=OIsX}           % Outcome
     S          =  O SUCHTHAT (\ o | ~ o.isX)           % State
     ATr        =  (S, O) -> Bool                       % Atomic Transition

CONST
     letter     := "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".set
     digit      := "0123456789".set
     ids        := {id | true}
     globals    := {id1, id2 | id1 + "." + id2}
     noX        := ""
     retX       := "$ret"
     loopX      := "$loop"
     typeX      := "$type error"
     trueV      :  V                                    % the value true

FUNC OIsX(o) -> Bool = RET o("$x") # noX                % o.isX
```

To write the meaning functions we need types for the representations of the main nonterminals of the language: `id, name, exceptionSet, type, exp, cmd, routineDecl, module`, and `program`. Rather than giving the detailed representation of these types or a complete set of operations for making and analyzing their values, we write `C« c1 [] c2 »` for a command composed from subcommands `c1` and `c2` with `[]`, and so forth for the rest of the command forms. Similarly we write `E« e1 + e2 »` and `R« FUNC Succ(x: INT)->INT = RET x+1 »` for the indicated expression and function, and so forth for the rest of the expression and routine forms. This notation makes the specification much more readable. `Id`, `Name`, and `XS` are declared above.

```
TYPE T          =  SET V                                % Type
     E          =  [...]                                % Expression
     C          =  [...]                                % Command
     R          =  [id, ...]                            % RoutineDecl
     Mod        =  [id, tops: SET TopLevel]             % Module
     TopLevel   =  (R + ...)                            % module toplevel decl
     Prog       =  [ms: SET Mod, ts: SET TopLevel]      % Program
```

The meaning of an `id` or `var` is just the string, of an `exceptionSet` the set of strings that are the exceptions in the set, of a `type` the set of values of the type. For the other constructs there are meaning functions defined below: `ME` for expressions and `MC` and `MR` for atomic commands and routines. The meaning functions for `module`, `toplevel`, and `program` are left as exercises.

Expressions

An expression maps a state to a value or exception. Evaluating an expression does not change the state. Thus the meaning of expressions is given by a partial function `ME` with type `E->S->(V + X)`; that is, given an expression, `ME` returns a function from states `S` to results (values `V` or exceptions `X`). `ME` is defined informally for all of the expression forms in section 5 of the reference manual. The possible expression forms are literal, variable, and invocation. We give formal definitions only for invocations and `LAMBDA` literals; they are written in terms of the meaning of commands, so we postpone them to the next section

Type checking

For type checking to work we need to ensure that the value of an expression always has the type of the expression. We do this by structural induction, considering each kind of expression. The type checking of return values ensures that the result of an invocation will have its declared type. Literals are trivial, and the only other expression form is a variable. A variable declared with `VAR` is initialized to a value of its type. A formal parameter of a routine is initialized to an actual by an invocation, and the type checking of arguments (see `MR` below) ensures that this is a value of the variable's type. The value of a variable can only be changed by assignment.

An assignment `var := e` requires that the value of `e` have the type of `var`. If the type of `e` is not equal to the type of `var` because it involves a union or a `SUCHTHAT`, this check can't be done statically. To take account of this and to ensure that the meaning of expressions is independent of the static type checking, we assume that in the context `var := e` the expression `e` is replaced by `e AS t`, where `t` is the declared type of `var`. The meaning of `e AS t` in state `s` is `ME(e)(s)` if that is in `t` (the set of values of type `t`), and the exception `typeX` otherwise; this exception can't be handled because it is not named by an identifier and is therefore a fatal error.

We do not give a practical implementation of the type check itself, that is, the check that a value actually is a member of the set of values of a given type. Such an implementation would require too many details about how values are represented. Note that what many people mean by "type checking" is a proof that every expression in a program always has a result of the correct type. This kind of completely static type checking is not possible for Spec; the presence of unions and `SUCHTHAT` makes it undecidable. Sections 4 and 5 of the reference manual define what it means for one type to fit another and for a type to be suitable. These definitions are a sketch of how to implement as much static type checking as Spec easily admits.

Atomic commands

An atomic command relates a state to an outcome; in other words, it is defined by an `ATr` (atomic transition) relation. Thus the meaning of commands is given by a function `MC` with type `C->ATr`, where `ATr = [s, o] -> Bool`. We can define the `ATr` relation for each command by a predicate: a command relates state `s` to outcome `o` iff the predicate on `s` and `o` is true. We give the predicates in the table on the next page and explain them in the text that follows the table; the predicates apply provided there are no exceptions.

Here are the details of how to handle exceptions and how to actually define the `MC` function. You might want to look at the predicates first, since the meat of the semantics is there.

Command	*Predicate*
`SKIP`	`o = s`
`HAVOC`	`true`
`RET e`	`o = s{"$x" -> retX, $a -> ME(e)(s)}`
`RET`	`o = s{"$x" -> retX}`
`RAISE id`	`o = s{"$x" -> "id"}`
`e1(e2)`	`( EXISTS r: Routine | r = ME(e1)(s) /\ r(s{"$a" -> ME(e2)(s)}, o) )`
`var := e` [1]	`o = s{var -> ME(e)(s)}`
`var := e1(e2)` [1]	`MC(C« e1(e2); var := $a »)(s, o)`
`e => c0`	`ME(e)(s) = trueV /\ MC(c0)(s, o)`
`c1 [] c2`	`MC(c1)(s, o) \/ MC(c2)(s, o)`
`c1 [*] c2`	`MC(c1)(s, o) \/ ( MC(c2)(s, o) /\ ~(EXISTS o' | MC(c1)(s, o')) )`
`c1 ; c2`	`MC(c1)(s, o) /\ o .isX \/ ( EXISTS o' | MC(c1)(s, o') /\ ~ o'.isX /\ MC(c2)(o',o ) )`
`c1 EXCEPT xs => c2`	`MC(c1)(s, o) /\ ~ o ("$x") IN xs \/ ( EXISTS o' \| MC(c1)(s, o') /\ o'("$x") IN xs /\ MC(c2)(o'{"$x" -> noX}, o) )`
`VAR id: T | c0`	`( EXISTS v, o' | v IN T /\ MC(c0)(s {id -> v}, o') /\ o = o'{id -> } )`
`VAR id: T := e | c0`	`MC(C«VAR id: T | id = e => c0»)(s, o)`
`<< c0 >>`	`MC(c0)(s, o)`
`IF c0 FI`	`MC(c0)(s, o)`
`BEGIN c0 END`	`MC(c0)(s, o)`

`DO c0 OD` is the fixed point of the equation `c = c0; c [*] SKIP`

[1] The first case for `assignment` applies only if the right side is not an `invocation` of an `APROC`. Because an invocation of an `APROC` can have side effects, it needs different treatment.

Table 1: The predicates that define `MC(command)(s, o)` when there are no exceptions raised by expressions at the top level in `command`.

The table of predicates has been simplified by omitting the boilerplate needed to take account of `$havoc` and of the possibility that an expression is undefined or yields an exception. If a command containing expressions `e1` and `e2` has predicate `P` in the table, the full predicate for the command is

```
      s("$havoc")                                  % anything if $havoc
   \/    ME(e1)!s /\ ME(e2)!s                      % no outcome if undefined
      /\ (   ME(e1)(s) IS V /\ ME(e2)(s) IS V /\ P
          \/ ME(e1)(s) IS X /\ o = s{ "$x" -> ME(e1)(s) }
          \/ ME(e2)(s) IS X /\ o = s{ "$x" -> ME(e2)(s) } )
```

If the command contains only one expression `e1`, drop the terms containing `e2`. If it contains no expressions, the full predicate is just the predicate in the table.

Once we have the full predicates, it is simple to give the definition of the function `MC`. It has the form

```
FUNC MC(c) -> ATr =
    IF
    ...
    []  VAR var, e | c = «var := e» =>
        RET (\ o, s | full predicate for this case )
    ...
    []  VAR c1, c2 | c = «c1  ; c2» =>
        RET (\ o, s | full predicate for this case )
    ...
    FI
```

First we do the simple commands, which don't have subcommands. All of these that don't involve an invocation of an `APROC` are deterministic; in other words, the relation is a function. Furthermore, they are all total unless they involve an invocation that is partial.

A `RET` produces the exception `retX` and leaves the returned value in `$a`.

A `RAISE` yields an exceptional outcome which records the exception `id` in `$x`.

An `invocation` relates `s` to `o` iff the routine which is the value of `e1` (produced by `ME(e1)(s)`) does so after `s` is modified to bind `"$a"` to the actual argument; thus `$a` is used to communicate the value of the actual to the routine.

An `assignment` leaves the state unchanged except for the variable denoted by the left side, which gets the value denoted by the right side. Recall that assignment to a component of a function, sequence, or record variable is shorthand for assignment of a suitable constructor to the entire variable, as described in the reference manual. If the right side is an `invocation` of a procedure, the value assigned is the value of `$a` in the outcome of the invocation; thus `$a` also communicates the result of the invocation back to the invoker.

Now for the compound commands; their meaning is defined in terms of the meaning of their subcommands.

A guarded command `e => c` has the same meaning as `c` except that `e` must be true.

A choice relates `s` to `o` if either part does.

An else `c1 [*] c2` relates `s` to `o` if `c1` does or if `c1` has no outcome and `c2` does.

A sequential composition `c1 ; c2` relates `s` to `o` if there is a suitable intermediate state, or if `o` is an exceptional outcome of `c1`.

`c1 EXCEPT xs=>c2` is the same as `c1` for a normal outcome or an exceptional outcome not in the exception set `xs`. For an exceptional outcome `o'` in `xs`, `c2` must relate `o'` as a normal state to `o`. This is the dual of the meaning of `c1 ; c2` if `xs` includes all exceptions.

`VAR id: t | c` relates `s` to `o` if there is a value `v` of type `t` such that `c` relates (`s` with `id` bound to `v`) to an `o'` which is the same as `o` except that `id` is undefined in `o`. It is this existential quantifier that makes the specification useless as an interpreter for Spec.

`<< ... >>`, `IF ... FI` or `BEGIN ... END` brackets don't affect `MC`.

The meaning of `DO c OD` can't be given so easily. It is the fixed point of the sequence of longer and longer repetitions of `c`.[2] It is possible for `DO c OD` to loop indefinitely; in this case it relates `s` to `s` with `"$x"->loopX`. This is not the same as relating `s` to no outcome, as `false => SKIP` does.

The multiple occurrences of `declInit` and `var` in `VAR declInit*` and `(varList):=exp` are left as boring exercises, along with routines that have several formals.

Routines

Now for the meaning of a routine. We define a meaning function `MR` for a `routineDecl` that relates the meaning of the routine to the meaning of the routine's body; since the body is a command, we can get its meaning from `MC`. The idea is that the meaning of the routine should be a relation of states to outcomes just like the meaning of a command. In this relation, the pseudo-name `$a` holds the argument in the initial state and the result in the outcome. For technical reasons, however, we define `MR` to yield not an `ATr`, but an `S->ATr`; a local state (`static` below) must be supplied to get the transition relation for the routine. For a `LAMBDA` this local state is the current state of its containing command. For a routine declared at top level in a module this state is empty.

The `MR` function works in the obvious way:

1. Check that the argument value in `$a` has the type of the formal.
2. Remove local names from the state, since a routine shares only global state with its invoker.
3. Bind the value to the formal.

[2] For the details of this construction see G. Nelson, A generalization of Dijkstra's calculus, *ACM Trans. Programming Languages and Systems* **11**, 4, Oct. 1089, pp 517-562.

4. Find out using MC how the routine body relates the resulting state to an outcome.
5. Make the invoker's outcome from the invoker's local state and the routine's final global state.
6. Deal with the various exceptions in that outcome.

   A retX outcome results in a normal outcome for the invocation if the result has the result type of the routine, and a typeX outcome otherwise.

   A normal outcome is converted to typeX.

   An exception raised in the body is passed on.

```
FUNC MR(r) -> (S->ATr) = VAR id1, id2, t1, t2, xs, c0 |
       r = R« APROC id1(id2: t1)->t2 RAISES xs = << c0 >> »
    \/ r = R« FUNC  id1(id2: t1)->t2 RAISES xs =    c0    » =>
    RET (\ static: S | (\ s, o |
         s("$a") IN t1                              % if argument typechecks
      /\ ( EXISTS g: S, s', o' |
              g = s.restrict(globals)               % g is the current globals
           /\ s' = (static + g){id2 -> s("$a")}     % s' is initial state for c0
           /\ MC(c0)(s', o' )                       % apply c0
           /\ o = (s + o'.restrict(globals))        % restore old locals from s
                {"$x" ->                            % adjust $x in the outcome
                  (    o'("$x") = retX =>
                         (    o'("$a") IN t2 => noX % retX means normal outcome
                           [*] typeX )              % if result typechecks;
                     [*] o'("$x") = noX => typeX    % normal outcome means typeX;
                     [*] o'("$x)                    % pass on exceptions
                  )
                }
    \/ ~ s("$a") IN t1 /\ o = s{"$x" -> typeX}      % argument doesn't typecheck
    ) )                                             % end of the two lambdas
```

We leave the meaning of a routine with no result as an exercise.

Invocation and LAMBDA expressions

We have already given in MC the meaning of invocations in commands, so we can use MC to deal with invocations in expressions. Here is the fragment of the definition of ME that deals with an E that is an invocation e1(e2) of a function. It is written in terms of the meaning MC(C«e1(e2)») of the invocation as a command, which is defined above. The meaning of the command is an atomic transition aTr, a predicate on an initial state and an outcome of the routine. In the outcome the value of the pseudo-name $a is the value returned by the function. The definition given here discards any side-effects of the function; in fact, in a legal Spec program there can be no side-effects, since functions are not allowed to assign to non-local variables or call procedures.


```
FUNC ME(e) -> (S -> (V + X)) =
     IF
     ...
     [] VAR e1, e2 | e = E« e1(e2) » =>
     % if E is an invocation its meaning is this function from states to values
          VAR aTr := MC(C« e1(e2) ») |
            RET ( LAMBDA (s) -> V =
                % the command must have a unique outcome, that is, aTr must be a
                % function at s. See Relation in section 9 of the reference manual
                VAR o := aTr.func(s) | RET (~o.isX => o("$a") [*] o("$x")) )
     ...
     FI
```

The result of the expression is the value of $a in the outcome if it is normal, the value of $x if it is exceptional. If the invocation has no outcome or more than one outcome, ME(e)(s) is undefined.

The fragment of ME for LAMBDA uses MR to get the meaning of a FUNC with the same signature and body. As we explained earlier, this meaning is a function from a state to a transition function, and it is the value of ME((LAMBDA ...)). The value of (LAMBDA ...), like the value of any expression, is the result of evaluating ME((LAMBDA ...)) on the current state. This yields a transition function as we expect, and that function captures the local state of the LAMBDA expression; this is standard static scoping. .

```
     IF
     ...
     [] VAR signature, c0 | e = E« (LAMBDA signature = c0) » =>
          RET MR(R« FUNC id1 signature = c0 »)
     ...
     FI
```