6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 5: September 28, 1992

Lecturer: Butler Lampson *Scribe: Mitchell Charity*[1]

1 Handouts

Handout 13 - Rudiments of Logic
Handout 14 - Atomic Semantics of Spec
Handout 15 - Solutions to Problem Set #1
Handout 16 - History Variables
Lecture Notes #3.

2 Redo Logs (cont)

Continued from last Wednesday (Lecture #4, September 21).

Use redo logs to implement arbitrary updates which are atomic in the presence of crashes.

2.1 Specification of Sequential Transactions

Abstract variables:
vs volatile state - erased by crash
ss stable state - unaffected by crash

Abstract operations:

```
Do(a)    = << vs := a(vs) >>
Commit   = << ss := vs >>
Crash    = << vs  := ss >>
```

Doing (applying) an action a changes the volatile stable vs.
When you Commit, the stable state gets set, atomically, to the volatile state.
Thus, one can do lots of individual things (actions), one at a time, and then commit, and ss will change atomically.
Crash forgets the volatile state, and thus is, in some sense, the opposite of Commit.
(For simplicity, this description leaves out both value returns and the "meaning" of actions.)

[1]These notes take the form of a dialogue between the lecturer and the class.

This is a completely general mechanism, given arbitrary actions a, of implementing crash atomic updates of arbitrary complexity.

2.2 Implementation Using Redo Log

The implementation idea is to keep the implementation's stable state in some stable place, and to have two logs - a volatile log and a stable log.

```
VAR ss    implementation's stable state
    vl    implementation's volatile log
    sl    implementation's stable log
```

A log is a sequence of updates. An update is such that it can be atomically applied to the stable state.
The abstraction function is:

```
AF  ss = ss + sl
    vs = ss + sl + vl                    (this, not the handout, is correct)
```

The first just means the abstract stable state is implementation's stable state with the application of the implementation's stable log.
(Here, the $+$ operator sequentially applies, to a state, each one of the updates in a log.)

How do we implement the log operations?

Do:

```
Do(a) =  vl := vl + Updates(a)
```

Do appends the new action a to the volatile log.
Implementationally, it appends the updates of a, rather than a as one big chunk.
As noted, the idea behind Updates is to break down the action into small enough things that each one can be done atomically. So if a is complex, it may have to be broken up quite a bit.

For example, if the stable state is implemented on the disk, each update must be small enough that it can implemented in a single disk write. If the action is to write a megabyte, and the disk blocks are a kilobyte, a will have to be broken down into a thousand updates. Of course, we can code the updates in some clever way, but logically there will have to be the thousand updates, because we can only write one disk block at a time atomically.

This implementation of Do correctly updates the volatile state, as:

```
vs = ss + sl + vl
```

Commit:

```
Commit =  sl := sl + vl
```

An interesting note is that this doesn't change the volatile log, so the volatile state is now the stable log followed by two copies of the volatile log.

```
vs := ss + (sl_original + vl) + vl
```

This doesn't matter given the *log idempotency* property (ie it is ok to repeat stuff at the end of the log).

There is more in the handout, but this is sufficient to do the job.

Crash:

```
Crash =  vl := {}
```

This corresponds to setting vs to ss:

```
vs_abstract = ss_impl + sl_impl + vl_impl
vs_abstract = ss_abstract       + vl_impl
vs_abstract = ss_abstract,   when vl_impl = {}
```

So we have (sortof) succeeded in implementing the redo log abstraction.

2.3 Redo Log Implementation - atomicity

But what is weird about this?
The implementations of Do, Commit, and Crash should be atomic, because the abstract versions are atomic, and we want everything to be straight forward.

So let's look at each again:

Do just changes volatile state, and there is no difficulty in making that atomic (given that there is no concurrency).

Commit is less obvious because it adds a potentially large thing to the stable log, which could involve a number of disk writes.
So making this an atomic operation will not be completely trivial.
But is much simpler than the original problem, which was to implement arbitrary actions. Now instead, we need to implement the process of appending an arbitrary large pile of update to the end of the stable log.
There are many possible ways to do that.

One way is described in the problem set 2.
The trick is, when writing something into the log, you arrange that the region of disk containing the log has the property that every disk block starts with a recognizable tag, set for example to 0. You then write blocks in in which the first byte is always set to 1. That allows you to tell how much writing has actually been done, when you come back to look at the log later. Thats the essential idea. It is part of the problem set to write down the implementation and abstraction function and so forth.

2.4 Redo Log Implementation - need explicit volatile state

Now we have to go back and correct a simplification we made earlier.
We left out that Do might return something. An action not only modifies state, but may also return some value from the call.

The previous specification of Do:

```
Do(a) = << vs := a(vs) >>
```

is extended to have a return value v:

```
VAR v | << (vs,v) := a(vs); RET v >>
```

As we need the return value, Do actually needs to apply a.
So we need a vs to apply it to.
But we don't have one. We have not been keeping one.

This is flaky if you have a lot of actions that return values.
Every time the client does a Do, you have to laboriously reconstruct enough of the volatile state, by applying the current volatile log, to determine the return value.
That is not very attractive.

So, in most cases, we add another component, the vs, to the state variables in the implementation. With it we explicitly track the changes in the vs as opposed to just representing them implicitly, in the vl, as a sequence of updates.
So our implementation state variables are:

```
VAR ss
    vl
    sl
    vs                    (added)
```

For a concrete example, think of a filesystem.
The reads have to be affected by the writes which are done. If I do a write 6 to byte 1000 and then read byte 1000, I have to get 6 back, not the previous value.
So if all I have done is remembered in the log that I have written 6 to byte 1000, then when I do the read I have to search the log to see whether there are any writes in the log which should affect the answer.
An that, in general, is going to be somewhat painful and expensive.

So usually you don't do that.
And we will see, in about 3 or 4 weeks, a considerably more complicated but also much more practical strategy for maintaining the volatile state efficiently. The difficulty with maintaining the volatile state is that it might get very big - you might run out of memory to store it. That gives rise to a number of complications that we will be discussing in a few weeks, but for now, we are not going to go into that.

2.5 Redo Log Implementation - Redo

There is another difficulty.
Each Commit adds to the stable log, which keeps getting longer and longer.
Further, nothing in our implementation every changes the stable state.

So, not only do we have this problem that we are not actually representing the volatile state, we are not really representing the stable state either.
There is a a stable state variable, but having never changed it, we might as well not have written it.
As more and more work gets done, there gets to be more and more log to search through in order to find out what is going on.

Of course, the only reason to find out what is going on is if you have to return an answer, but the system is not very interesting if it is write only. There are going to be reads occasionally. In fact, usually there are more reads than writes.

So clearly we have to do something about the stable state.
But what are we going to do?

We add a new operation Redo.
Redo essentially takes things, one at a time, out of the stable log, and applies them to the stable state.

Illustration:
Given a stable log with several updates, sl = $\{$a, b, c$\}$, Redo does:

```
ss := a(ss)
ss := b(ss)
ss := c(ss)
```

```
 sl := {}
```

Redo applies the updates in the stable log to the stable state.
Of course, it is possible to fail and crash in between these. So in order to guarantee that Redo gets to run to completion after a Commit, Crash, as well as Commit, runs Redo. So either way, we are going to keep going thru this whole sequence of applications until we succeed. Log idempotence permits us not to care if this happens repeatedly.

```
Commit = << sl := sl + vl >> ; Redo()

Crash  = << vl := {} >> ; Redo()
```

When we are all finished, we can set the sl to $\{\}$, because we don't need it once it has been applied.

The whole picture:
To implement arbitrary updates, and make them crash atomic, you have this abstraction with Do and Commit as the client actions, along with an uncontrollable action, Crash. You keep a log in which you record the actions. It is volatile as long as you just keep doing Do's. When you do a Commit, the log is made stable atomically. Then you do this redo thing which goes thru and actually executes these actions on the stable state. If you succeed in getting all the way to the end of that, then you reset the log. Then you are ready for another sequence of this whole process.

It should be pretty obvious that you don't have to arrange Redo in the a rigid way described here and in the handout. We do an exhaustive Redo as part of Commit. But we could do any prefix of the log, then delete that prefix, and then keep going. We actually have a lot of flexibility about the order in which we do the stable state update.

There are lots of practical reasons why you might want to take advantage of that flexibility, as the problem set suggests. You may wish to batch updates, as they require access to the disk, which is going to be slow. Further, as it is written, Commit requires the client to wait around while the relatively expensive stable update (Redo) operation takes place. There is no need for this. All that is required is to wait until the stable log has been changed. Having changed the stable log, the abstract stable state has been changed. Which is all you need to do.

Notice that this Redo operation has the property that it doesn't change the abstract state at all. All it is doing is applying the implementation's stable log to the implementation's stable state, but we already have it applied in the abstraction. The abstract stable state is defined as the concrete stable state with the concrete stable log applied ($ss_{abs} := ss_{impl} + sl_{impl}$). But that is all Redo does. And we have already "done" it in the abstraction function.

So as we execute redo, the concrete stable state is going to change, but because of the log idempotence property, redoing the log on the changed stable state is not going to have any abstract effect. To make this more concrete: the simplest actions to understand which have the idempotence property are straightforward writes to fixed addresses. You can repeat these as many times as you want, and you will still get the same answer as long as you finish up by writing them in the right order.

This set of ideas is used repeatedly in the course and in the construction of real systems which have fault tolerance. It is important to understand them.

2.6 Idempotence and arbitrary updates

Here is a general recipe for taking a arbitrary update and making it have the log idempotence property. It involves attaching uids to the updates.

The whole idea of the redo is to break the actions into atomic stable state updates and do them one at a time. As we have been doing previously, we will consider disk page writes to be the primitive atomic operation. However this need not be the case, we might be using some higher level abstraction than the disk, but it is convenient and very concrete.

The idea of this general technique for turning an arbitrary disk page update into one that has log idempotence, is to put a version stamp (or time stamp, or unique id), on the disk page, and put the same uid on the update. That way you can tell if the update has been applied.

Let's look at a concrete example.
Suppose that the disk page contains just one major variable x. Suppose the update is add 3 to x.

Of course real disk pages don't contain just one anything, they contain all kinds of stuff. The variable x just stands for whatever arbitrary pile of garbage is on the disk block.

This update obviously does not have the log idempotence property - if we do it twice we don't get the same answer we would get if we only do it once. In fact, most interesting updates do not have the log idempotence property. Thats why this is an interesting construction.

The notion is to replace the original update, with one which is conditional on the version of the disk page. We go to the page and find out its current version number, and then create an update which only acts if the version number is the same as it was when the update was created. When it acts, it does the original update, with an incremented version number.

```
             Original                Idempotent

The disk:      x: Int                   x: Int
                                        vn: Int

The update:    x := x+1                 vn = 10 => x  := x+1;
                                                   vn := 11
                                        [*] SKIP
```

So now, if the update has already been applied, it wont be done again because the version number wont be 10. It will be at least 11. And if subsequent updates come along, they will always increment the version number.
On the other hand, if it hasn't been applied yet, then the version number will be 10, and we will apply it.

Why will the version number be 10? Why wont it be 9, 8, or 7? Because the only way it could have gotten it up to being 10 at the time we generated this update, is that previously, we had updates which incremented it up thru 7 8 and 9. Those updates preceded this one in the log, because we put updates in the log in the same order that we generate them. Therefore, we definitely must be up to 10, if we replay the log up to this point.

This is a completely general method, using version numbers, or time stamps or unique ids, or whatever you want to call them, for taking an arbitrary update, and making it have the log idempotence property.

Question: Does it have to be one disk page at a time?

Yes. In this example, yes. The reason it has to be one disk page at a time, is because redo relies on having each individual update be atomic, and if we are building right on top of the raw disk, the only atomic operation we have is to write one disk page. Now, of course, we could build on top of some other abstraction, one that gives you bigger atomic operations. But what is written in the handout is extremely generic. All it says is that you have some way of representing a stable state. It doesn't say anything about disk pages. I have been using the disk pages in the filesystem as a concrete example but the idea is extremely general.

For example, it is very common to use these methods on top of a standard unix filesystem. If you want to build a database on top of a filesystem that can tolerate crashes, this is the standard way to do it. You represent the data in a file, you represent the updates in a log, which is another file, that you just append to. You then apply the atomic update operation, or whatever your filesystem gives you as atomic operations, depending what sort of filesystem you have. Though you have to read the manual pretty carefully in order to get even a clue as to what that might be half of the time. But it could be much more than single disk pages.

Question: The handout uses sets of uids rather than version numbers.

Our approach here is a bit cheaper to implement, as you don't need to maintain sets of uids. Another way to look at it is that this is an implementation of the slightly more abstract version with the sets of uids. This one is perhaps a little bit too detailed. If you look at real implementations, they don't necessarily increment the version number by 1, for example. What is very common to do in database systems is to actually use the position of the update in the log as the version number.
I gave this example to be very, very concrete, but this is the standard technique to make database

systems work, and there are a number of different strategies for choosing the version numbers. I don't think anyone uses exactly this strategy. Maybe some do. Certainly most systems don't use exactly this strategy.

Question: I don't see how this gives you fault tolerance.

What this does is provide the property that, if there is a crash, then anything that got done since the last commit is going to get lost. This is because what happens when there is a crash in the abstraction is that the volatile state gets reset to the stable state. The stable state only gets changed when there is a commit.

Question (cont): So there is still data lost...

There is data lost, but it is not arbitrary what gets lost. The client knows that after a failure, the thing is always in the state it was in after the last commit. Which is much much better than having done a whole bunch of updates and having no idea which ones got applied. Thats pretty hopeless. This is called crash atomicity, because it allows you to take a bunch of updates, group them together and then guarantee that either all of the updates in the group get done, or none of them get done.

Question: If you committed your updates to the stable log, each time you changed the stable log, then you would have fault tolerance.

In general, you don't want to do that.
Remember the example of maintaining the free list for the filesystem. We had two updates there, we had to add a block to the indexing structure at the bottom, and to turn off the bit in the bit table representing the free space. We didnt want to do only one of those, we wanted to guarantee that we got either both of them or neither. It would be a big loss if you said you _had_ to make each update stable.

It is a very important property of this that you get to choose when the thing becomes stable, because if you are in the middle of doing the updates, then the fact that you are still computing is encoding the fact that there is still more work to be done. But after a failure that information is all lost. So you have to have some other way to capture what constitutes a good state, as opposed to some intermediate state which still needs work. That is the function of commit. When I commit, that means now it is a good state. I am willing to live with this state if there is a failure.
In the example of filesystem, I would not be willing to live with a state in which I had allocated a page, but had not yet put it in the indexing structure, so it would be lost.

Question: What if we fail in the middle of a Redo operation?

Redo will get restarted because Crash will do Redo again.
Where will it redo from?

It will redo from whatever is left in the stable log. In this version of Redo which erases the stable log each time it is called, it would be from the most recent commit. But as I pointed out, it doesn't have to be like that. You could have a lot more flexibility about exactly when to take stuff out of the stable log.

3 Atomic Semantics of Spec

The purpose of the next hour is to make it clear that there is not any arm waving in the spec notation which we have given you. A translation of what I just said into fancy words is that I am going to give you a formal semantics of the spec language.

Now that is a formidable sounding term, and if you go 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 fix points and stuff like that. You are not going to hear anything like that in the next hour. 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 purpose of this is to make sure that there is no doubt that we really do know what we are talking about. And, in general, notations or facilities that exist for describing what computer systems should do, are not in that state of grace.

If you read the pascal manual or the C manual you will come away, if you read it carefully, with a number of questions, about what exactly happens if I do this and this. Questions which the manual will not answer adequately. Questions with the property that two reasonably intelligent people who have studied the thing carefully can come to different conclusions. And argue for a long time and not be able to decide what is the right answer by reading the manual.

There is of course one class of mechanisms for saying what the computer should do that tends not to have that property, and that is the machines instruction set. The reason for this is that it really tends to just be a garden variety state machine, with fairly simple transitions, which is not beyond the power of the guy who is writing the manual to describe properly. Whereas a programming language is not like that. It has much more power and generality, and wonderfulness, and there tends to be a lot of confusion there.

One of the things we want to be able to do is to write specifications for computer systems, and one example of a computer system is 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 which we have been looking at in the last week, which has been filesystems. For this purpose, spec is not the ideal descriptive notation. If you are in the business of writing the semantics of programming languages you wouldnt use spec. People have invented a large number of other notations and some of them are actually better than spec, although most of them are far worse. But it 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 trying to pick up a new one each time. For there are a lot of pitfalls involved in devising a new notation. So we are going to stick with this one and see what we can get out of it.

So those are the two things which will be going on for the next hour. 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 a pretty different kind of system than a filesystem.

For this lecture, we will only talk about the sequential semantics of Spec, the atomic semantics of Spec. We are not going to talk 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. Bill we be talking next week about the semantics of concurrency, which is unavoidably more complicated.

In a sequential spec, there are basically three constructs:

```
- Expressions
- Commands
- Routines
```

(I may occasionally accidentally substitute “statement” for “command”.)

In order to describe what each of these things mean, you first of all have to have some notion of what kind of thing the meaning of an expression or command might be. And then we have to in detail explain exactly what the meaning of each possible kind of expression is.

The basic technique we are going to use for that is the standard method you always use in a situation where you have things which are made up out of smaller things. That is called structural induction.

3.1 Structural Induction

The basic idea is, if you have something which is made up of an A and a B, and you know the meaning of each, and have 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
```

3.2 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 this is all there is, 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. So somehow the meaning we give to this whole construct has to express that.

Then 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 which changes the state, so that no matter what the x was before, it is 1 afterwards. Because thats what we want this assignment to mean.

Spec has been substantially simplified since last year. In particular, "references" (like pointers in C) have been ripped out. They complicate understanding of what is going on quite a bit, and they are not really necessary. When you are doing problems, if you feel the urge to call malloc, the correct thing to do is to make a sequence of whatever sort of thing you want to allocate, and then choose a new index in the sequence which isn't being used already. Make a sequence or make a function which maps an integer or some convenient sort of name into the values that you want to allocate.

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

3.2.1 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
    x := 3              a.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, one of the things you have to do is go through the program first and every place you see an x, if that x was declared in the current module a, then you replace in with a.x. You convert all references to global variables to these two part names. This makes things simpler, but crufty to read.

All the global state is going to consist of 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. The general parameters for the design of spec were to try to make it suitable for the kinds of specs and implementations that we do in this course, some of which are moderately complex.

Spec is plenty good enough for everything we need to do in this course. In general the rule was, if it's not really needed to write the specs of the kind we are doing in this course, then we left it out, to keep it simpler.

3.3 Expressions

What should the meaning of an expression be?

Note that expressions do *not* affect state.

Question: What about assignments?
Assignments are not expressions.
If you have been programming in C, you have this 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

```
- Expression  S -> V
```

An expression is a function of state to value.
It can be a partial function, as spec does not require that all expressions be defined.
But it has to be a function - we do require that everything is deterministic.

We want determinism so something like

```
f(x) = f(x)
```

always comes out true. Otherwise, life is just too confusing. If you want to do reasoning, and this isn't true, then life is just too hard. If a function is nondeterministic then obviously this neednt 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 3 types of expressions:

```
constant                1    lambda s -> V = RET 1
variable                x    lambda s -> V = RET s("x")
function application  f(x)
```

(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 could not write just RET 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 application. We will only talk about functions with a single argument. If you want a function with two arguments, you can make one by combining two functions of a single argument.

So what about x+y?
x+y is just T."+"(x,y)

Everything that is not a constant or a variable is an application. This should be a familiar concept for those of you who know SCHEME.

Semantics of function invocation:

So what are the semantics of function application?

Well, given a function T $->$ U, the correct type of its value 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 application: f(x)?
Remembering the rule of structural induction, what are we going to do? 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. Somehow we are going to need to get our hands on the meaning of f and the meaning of s, and then to put them together appropriately.
What is the meaning of f? s("f")

So,

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

How are we going to put it together, remembering the type f(x) is going to be?

f(x) λ **s = RET s("f") (s("x"),s)**

Now this could be complete nonsense. If s("f") evaluated to an integer, for instance. 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 apply 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 do you do about undefined functions?
Then the (T,S) $->$ U mapping is partial.

Question: Is f(x) = f(x) if f(x) is undefined?
No. I don't think it is the right thing to do to have it come out true in that situation. But those are deep waters and I propose to stay out of them.

3.4 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. What sort of thing is a command? 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 thing?

```
  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 that I have used is:

```
 - Commands     (S,S) -> Bool
```

Now, there is going to be a small complication because spec has exceptions. Which are pretty useful for writing many kinds of specifications. So we have to deal with the possibility that the result of a command is not a garden variety state, but this exception thing.

So we are going make a slight extension, have 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 that I have chosen that an outcome has the same type as a state, a function of names to values. We are going to throw in a couple of funny names, that you can't actually write in the program.
One is $x.
If o("$x") $->$ "" (empty string), then o is a garden variety state.
If o("$x") $->$ "exception-name", then there is an exception.

Most of the statements of spec do something different if one of their pieces produces an exception, than if none do. So we have to look at this component in the semantics.

Now we just work our way through the command constructs.

3.4.1 Commands - assignment

x := 1, or in general:

```
variable := expression
```

What we have to come up with is one of these expressions which is

```
lambda s,o = RET  ...
```

How is o going to be related to s? By the function returning true. This is a slight uncomfortable way to write this. If we choose this way of describing a relation, we have to come up with a predicate on the state and the outcome, which is true exactly when we want to make a transition. There is not going to be any exceptional outcome. We are encoding a relation between states. We are encoding the relations as these functions from a state and outcome to a bool. And the function is supposed to give back true, exactly when the relation holds.

So when does the relation hold for variable gets expression?

Well, perhaps when o(x) = exp?

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

ME is the meaning function for expressions.

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 = 1
      ->
           y = 94
```

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{"var":= ME(e)(s)}
```

This is just a function constructor, as in f{arg:=value}.

3.4.2 Commands - an alternate encoding

As we said before, there are many ways to code the command relation. One way we mentioned was:

```
 - Commands    S -> SET S
```

I tried that when writing the semantics for spec, but, either I did something stupid, or things just come out cruftier. But it certainly works.

There is another way to write this, which has a lot of advantages. That is to 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 this

```
(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 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 which 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 these y'=y stuff clutter things up an awful lot.

This was just an aside, but sometimes I will take advantage of this observation, and when talking generally about what kinds of things can go on in a specification, I will be willing to describe it using predicates, rather than using these functions of states to bool.

3.4.3 Commands - routine invocation f(x)

What are the semantics of routine application/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 here, and in the handout, is to use another pseudo-variable, $a. $a is used to pass the argument and get back the result.

The meaning of f(e) is going to be

```
f(e)
  lambda s,o = RET           s                                | Take the state,
                              {"$a":=ME(e)(s)}                | append the argument,
                    ME(f)(s)                                  | get the function,
                            (                   ,o)           | and apply it.

  lambda s,o = RET ME(f)(s)(s{"$a":=ME(e)(s)},o)
```

So 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 function relates that state to the outcome.

Another way of writing this, which isn't so nested, and might be clearer, would be to introduce some intermediate state s':

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

These two are exactly the same thing. The invocation relates s to o, iff the function 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 function.

Question: Why use ME(f)(s) rather than MR?
I would use MR if I was looking at a FUNC. But f is just a variable (which had better be bound to a function, or this wont typecheck).

An alternate encoding

Here is a different way of communicating the argument value to the function. You could take the view, that when I write the procedure

```
  PROC p (i: Int)
```

what we are doing 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(f)(s) ( ME(e)(s) ) (s,o)
```

This says, first, get the meaning of f. f is not going to be 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 function, namely what that function does when given that particular argument value.
So we pass it the argument value, and apply the resulting transition.

Encodings contrasted

These two alternatives are based on different choices about how to code the meaning of procedures. If you code the meaning of a routine simply as a transition, that is spec picks up the argument value out of this 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 all 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.

So in $ME(f)(s)(ME(e)(s))(s,o)$, $ME(f)(s)$ 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, and they don't really make that much difference, but they can create a lot of confusion. Firstly, because a bad choice can leave you in a thicket of notation. And secondly, because you can get confused about what choice was made.

So, while this

```
RET ME(f)(s)(s{"$a":=ME(e)(s)},o)
```

and this,

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

are two ways of writing exactly the same thing, this,

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

is different, and only makes sense with a different choice about what the meaning of a function is.

So while I argue that $ME(f)(s)(ME(e)(s))(s, o)$ is more beautiful, I propose to use $lambdas, o = RET\ ME(f)(s)(s\{``\$a" := ME(e)(s)\}, o)$ because I think it 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.

3.4.4 Commands - HAVOC

λ s,o = RET true

3.4.5 Commands - SKIP

λ s,o = RET s = o

In the handout, what I have done for the commands is to notice that there is a lot of boilerplating - this $\lambda s, o = RET$ stuff, is always the same. So I have given a table which shows for each syntactic form, what goes after the RET.

Now for the compound commands.

3.4.6 Commands - c1 [] c2

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

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

3.4.7 Commands - c1 [*] c2

Well, it is clear we should begin with

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

But what next?

One possibility would be

```
~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.

So, the correct encoding is

```
MC(c1)(s,o) \/ (ALL o' | ~MC(c1)(s,o')) /\  MC(c2)(s,o) )
```

By the way,
I handed out a small handout (#13) which I labeled ”Rudiments of Logic”. This is in response to my discovering last year that there were people who did not know what De Morgan's laws were.

4 Next Week

Lecture #6 will finish up semantics by covering

```
Commands
  ;
  EXCEPT
  VAR
  DO

Routines
  RET
```

Scribe's afterward:
In case you are at all tempted to do your own lecture notes in this manner, note that this required something more than 30 hours. The time might better have been spent clearly and concisely organizing the key concepts. — mcharity

TA's afterward:
. . . not to mention the 6 hours it took me (in trying) to edit it. — umesh