Software Testing, Quality Assurance and Maintenance Winter 2025

Lecture 5 — January 20, 2025

Patrick Lam *version 3*

The big topic for today is semantics. In SE 212 you saw axiomatic semantics, which generally are used to prove program correctness. There are other ways of reasoning about programs, though, namely operational semantics and denotational semantics. In this coure we are going to use operational semantics, and it's going to be needed for some of the reasoning that we're going to do.

Defining WHILE, a simple imperative language

Languages are complicated. We're going to work on a language, which we'll call WHILE[1]. We've removed almost everything from it. And yet, it still shows the essential complications of reasoning about code. You had a similar language in SE212.

Here's an example program.

```
1   { p := 0; x := 1; n := 2 };
2   while x <= n do {
3     x := x + 1;
4     p := p + m
5   } ;
6   print_state
```

Note that ; separates statements. Contrast this to C, where you end statements with a ;.

Syntactic entities. The syntactic entities of WHILE are as follows. We first have terminals:

- $n \in \mathbb{Z}$: integers;
- $\mathsf{true}, \mathsf{false} \in B$: Booleans;
- $x, y \in L$: locations (program variables);

as well as non-terminals:

- $a \in \mathit{Aexp}$: arithmetic expressions;
- $b \in \mathit{Bexp}$: Boolean expressions; and,
- $c \in \mathit{Stmt}$: statements.

Terminals are leaves in the Abstract Syntax Tree, and completely defined by their tokens (rather than their relationships to other AST nodes). Non-terminals contain zero or more terminals in terms of parsing, and contain zero or more AST tokens, as determined by the grammar. The containment relationship is represented by AST edges.

[1] Sometimes a similar language is called IMP.

Arithmetic Expressions. We can now give a grammar for arithmetic expressions *Aexp*.

$$
\begin{array}{rcll}
e & ::= & n & (\text{for } n \in \mathbb{Z}) \\
 & | & x & (\text{for } x \in \mathbb{L}) \\
 & | & \texttt{-}e & \\
 & | & e_1 \ aop \ e_2 & \\
 & | & \texttt{(}e\texttt{)} & \\
 & & & \\
aop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} &
\end{array}
$$

Some notes:

- Like in Python, you don't have to declare variables before use.
- All variables are integer-typed. We have Boolean expressions, but not Boolean variables.
- Expressions have no side-effects—all effects are explicit.

Boolean Expressions. Also for boolean expressions *Bexp*.

$$
\begin{array}{rcll}
b & ::= & \texttt{true} & \\
 & | & \texttt{false} & \\
 & | & \texttt{not } b & \\
 & | & e_1 \ rop \ e_2 & (\text{for } e_1, e_2 \in Aexp) \\
 & | & b_1 \ bop \ b_2 & (\text{for } b_1, b_2 \in Bexp) \\
 & | & \texttt{(}b\texttt{)} & \\
 & & & \\
rop & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>=} \mid \texttt{>} & \\
bop & ::= & \texttt{and} \mid \texttt{or} &
\end{array}
$$

Statements. Finally, statements s.

$$
\begin{array}{rcl}
s & ::= & \texttt{skip} \\
 & | & x\texttt{:=}e \\
 & | & \texttt{if } b \texttt{ then } s \ [\texttt{else } s] \\
 & | & \texttt{while } b \texttt{ do } s \\
 & | & \{ \ slist \ \} \\
 & | & \texttt{print\_state} \\
 & | & \texttt{assert } b \mid \texttt{assume } b \mid \texttt{havoc } v_1, \ldots, v_N \\
slist & ::= & s \ (\texttt{;}s)^* \\
prog & ::= & slist
\end{array}
$$

More notes:

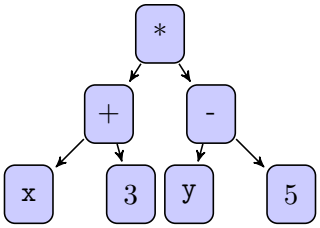
- The semicolon `;` separates statements, rather than terminating them.

- All side-effects in WHILE are caused by statements (not expressions).
- We've dropped many features from WHILE (references, function calls, ...), but it's still fiendishly hard to analyze!

Abstract Syntax Trees & Visitors

We've seen this before in Lecture 3, but the AST abstractly represents the source code in tree form; each node represents some syntactic construct from the code, e.g. statements, expressions, variables, operators. It's abstract because it drops the tokens like `(` as well as, for most langages, whitespace and comments. Order of operations is encoded in the tree edges.

Here's an AST for `(x + 3) * (y-5)`.



The slides contain content about language parsing, but I'll assume you know this from CS 241.

WHILE AST implementation in Python. Here is some code that implements a WHILE AST. It has one class per syntactic entity, one field per child, and the class hierarchy corresponds to the semantic hierarchy (i.e. the class encodes the type of node).

```
1  class Ast(object):
2    """Base class of AST hierarchy"""
3    pass
4
5  class Stmt(Ast):
6    """A single statement"""
7    pass
8
9  class AsgnStmt(Stmt):
10   """An assignment statement"""
11   def __init__(self, lhs, rhs):
12     self.lhs = lhs
13     self.rhs = rhs
14
15 class IfStmt(Stmt):
16   """If-then-else statement"""
17   def __init__(self, cond, then_stmt, else_stmt=None):
18     self.cond = cond
19     self.then_stmt = then_stmt
20     self.else_stmt = else_stmt
```

Visitor pattern. I expect you to know about the visitor pattern from CS 247. I'm just going to make two Python-specific comments:

- instead of using polymorphism to visit, as done in Java, we find the target using the class name as part of the method, e.g. `visit_Stmt()`;
- the `visit()` method of the Visitor uses reflection (it calls the method indicated by the string) rather than polymorphism to make the call, and hence we don't need `accept()`.

Here is an example Python visitor for WLANG:

```
1  class AstVisitor(object):
2      """Base class for AST visitor"""
3
4      def __init__(self):
5          pass
6
7      def visit(self, node, *args, **kwargs):
8          """Visit a node."""
9          method = "visit_" + node.__class__.__name__
10         visitor = getattr(self, method)
11         return visitor(node, *args, **kwargs)
12
13     def visit_BoolConst(self, node, *args, **kwargs):
14         visitor = getattr(self, "visit_" + Const.__name__)
15         return visitor(node, *args, **kwargs)
16
17         # etc...
```

and a visitor implementation:

```
1  class PrintVisitor(AstVisitor):
2      """A printing visitor (excerpts)"""
3
4      def visit_IntVar(self, node, *args, **kwargs):
5          self._write(node.name)
6
7      def visit_IntConst(self, node, *args, **kwargs):
8          self._write(node.val)
9
10     def visit_Exp(self, node, *args, **kwargs):
11         if node.is_unary():
12             self._write(node.op)
13             self.visit(node.arg(0))
14         else:
15             self._open_brkt(**kwargs)
16             self.visit(node.arg(0))
17             for a in node.args[1:]:
18                 self._write(" ")
```

```
19                  self._write(node.op)
20                  self._write(" ")
21                  self.visit(a)
22              self._close_brkt(**kwargs)
```

Exercise. Write two visitors that count the number of statements in a program.

1. the visitor should be stateless and return the number of statements;
2. the visitor uses internal state (a field) to keep track of the number of statements.

Here is a stateless visitor implementation. The number of statements gets passed up the tree, basically; at each node, the visitor at a node asks the children of that node about how many statements they have, and returns the childrens' result plus any statements in that node. The statements that have instructions are `StmtList`, `IfStmt`, `WhileStmt`, and `Stmt`.

```
1  class StmtCounterStateless(ast.AstVisitor):
2      def __init__(self):
3          super(StmtCounterStateless, self).__init__()
4
5      def visit_StmtList(self, node, *args, **kwargs):
6          if node.stmts is None:
7              return 0
8          res = 0
9          for s in node.stmts:
10             res = res + self.visit(s)
11         return res
12
13     def visit_IfStmt(self, node, *args, **kwargs):
14         res = 1 + self.visit(node.then_stmt)
15         if node.has_else():
16             res = res + self.visit(node.else_stmt)
17         return res
18
19     def visit_WhileStmt(self, node, *args, **kwargs):
20         return 1 + self.visit(node.body)
21
22     def visit_Stmt(self, node, *args, **kwargs):
23         return 1
```

And the stateful visitor is like this. There are more potential concurrency problems with a stateful visitor, but you're not passing information around.

```
1  class StmtCounterStatefull(ast.AstVisitor):
2      def __init__(self):
3          super(StmtCounterStatefull, self).__init__()
4          self._count = 0
5
```

```
6       def get_num_stmts(self):
7           return self._count
8
9       def count(self, node, *args, **kwargs):
10          self._count = 0
11          self.visit(node, *args, **kwargs)
12
13      def visit_StmtList(self, node, *args, **kwargs):
14          if node.stmts is None:
15              return
16          for s in node.stmts:
17              self.visit(s)
18
19      def visit_Stmt(self, node, *args, **kwargs):
20          self._count = self._count + 1
21
22      def visit_IfStmt(self, node, *args, **kwargs):
23          self.visit_Stmt(node)
24          self.visit(node.then_stmt)
25          if node.has_else():
26              self.visit(node.else_stmt)
27
28      def visit_WhileStmt(self, node, *args, **kwargs):
29          self.visit_Stmt(node)
30          self.visit(node.body)
```

Non-determinism vs randomness. Here's a side comment which is important to say somewhere, so we'll say it here.

A *deterministic* function always returns the same result on the same input, e.g. $f(5) = 10$. (You can prove that if $f(x) = y_1$ and $f(x) = y_2$, then $y_1 = y_2$).

A *non-deterministic* function may return different values on the same input, e.g. $g(5) \in [0, 10]$—that is, g returns a non-deterministic value between 0 and 10. In our context, we use nondeterminism to model the worst possible adversary/environment. These aren't functions in the mathematical sense—they are functions in the programming sense.

A *random* function may choose a different value with a probability distribution, e.g. $h(5)$ might yield 3 with probability 0.3; 4 with probability 0.2; and 5 with probability 0.5.

Semantics

This course relies on the semantics of programming languages, which is topic of CS 442. In SE 212, you used the axiomatic semantics for a language very much like WHILE to prove (by hand) that programs have certain behaviours. We're going to use Dafny to automate that in this course.

In this lecture, though, we are going to look at this course's tool, the *operational semantics*, a

different notation for expressing the semantics of a programming language.

For more about semantics, you can refer to the book:

Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. 2007.

Through the library, you can get an authenticated link from `https://uwaterloo.ca/library/make-link` to the book contents at `https://link.springer.com/book/10.1007/978-1-84628-692-6`. Chapters 1 and 2 are relevant here.

Just asking questions.

- What is the "meaning" of a given WHILE statement (or expression)?
- How do we evaluate WHILE statements and expressions?
- How are the evaluator and the meaning related?
- How can we reason about the effect of a command?

Kinds of semantics. You've seen axiomatic semantics. All semantics ascribes meaning to programs, but the tools are different, as is what is easy to do with each. Here are the three main kinds.

Axiomatic semantics: the meaning of a program is defined in terms of its effect on the truth of logical assertions. Used in Hoare Logic.

Denotational semantics: the meaning of a program is defined as the mathematical object it computes (e.g. partial functions). Abstract interpretation is based on denotational semantics.

Operational semantics: a step-by-step approach where the meaning of a program is defined by formalizing the individual computation steps of a program. We are going to see natural (big-step) semantics as well as structural (small-step) semantics here.

Semantic domain of WHILE. The semantics provides, for each statement, a transformation of the state of the program; and for each expression, a translation of that expression into a value.

So what is this program state?

A *state* q (also known as a store) is a function from variables/locations L to $\mathbb{Z}$; it assigns a value for every variable. We write $q(x)$ to denote the value of variable x in state q. We can express a state as a list of variable/value pairs, e.g. $[x := 10; y := 15]$ is a state with variables x and y.

The set of all states is Q.

Judgments (big-step/natural operational semantics). We write:

$$\langle e, q \rangle \Downarrow n$$

to mean that expression e in state q evaluates to value n[2].

[2] if you're looking at the book, it uses $\Rightarrow$ rather than $\Downarrow$.

This is a *judgment*. A judgment is a relation between expressions e, states q, and values n. Or, $\Downarrow$ is a function taking e and q and yielding q. Judgments give expressions meaning.

Next, we'll define $\Downarrow$, e.g. for things like

$$\langle e_1 + e_2, q\rangle \Downarrow n$$

We do this using inference rules, which you've already seen in SE 212.

$$\frac{F_1 \cdots F_n}{G}\ \mathrm{H}$$

Recall that the F_i are premises, G is the conclusion, and H is the side-condition. Side-conditions aren't used in this lecture, but come up later.

There are axioms, which are unconditionally true. In the derivation below, G doesn't require any premises or side-conditions.

$$\frac{}{G}$$

For an axiom, it's OK to omit the line separating (empty) premises and conclusion.

Inference rules. Again a refresher from SE 212. We define evaluation by using inference rules. An evaluation starts from axioms and reaches a conclusion about the value of an expression, or the final state of a program. Note that evaluations can get stuck; getting stuck, when there is no legal next inference rule to apply and yet evaluation is not finished, is how the semantics (implicitly) expresses errors.

Big-Step Operational Semantics

The *structural operational semantics* defines rules for language constructs. Usually one, but sometimes a few. Rules are defined based on the structure of the expressions.

We can now give semantics for WHILE, starting with arithmetic expressions, continuing with booleans, and then with statements.

Defining arithmetic expressions *Aexp*.

$$\frac{}{\langle n, q\rangle \Downarrow n} \qquad \frac{}{\langle x, q\rangle \Downarrow q(x)}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \quad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{+} e_2, q\rangle \Downarrow n_1 + n_2} \qquad \frac{\langle e_1, q\rangle \Downarrow n_1 \quad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{-} e_2, q\rangle \Downarrow n_1 - n_2}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \quad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{*} e_2, q\rangle \Downarrow n_1 * n_2}$$

Using inference rules: derivations. We paste together applications of the rules like this.

$$\dfrac{\dfrac{}{\langle 5, q\rangle \Downarrow 5} \qquad \dfrac{\dfrac{}{\langle 7, q\rangle \Downarrow 7} \qquad \dfrac{}{\langle 2, q\rangle \Downarrow 2}}{\langle \texttt{7*2}, q\rangle \Downarrow 14}}{\langle \texttt{5+(7*2)}, q\rangle \Downarrow 19}$$

It's a *derivation* if it's a well-formed application of inference rules. Derivations infer new facts from existing ones (starting at axioms).

Defining Boolean expressions *Bexp*. It turns out that in WHILE there is a nesting property where you can define arithmetic expressions first, and then Boolean expressions on top of arithmetic expressions. In some languages they depend on each other, making things more complicated.

$$\frac{}{\langle \mathsf{true}, q\rangle \Downarrow \mathsf{true}} \qquad \frac{}{\langle \mathsf{false}, q\rangle \Downarrow \mathsf{false}}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{ = } e_2, q\rangle \Downarrow n_1 = n_2} \qquad \frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{ <= } e_2, q\rangle \Downarrow n_1 \leq n_2}$$

$$\frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{ and } e_2, q\rangle \Downarrow n_1 \wedge n_2} \qquad \frac{\langle e_1, q\rangle \Downarrow n_1 \qquad \langle e_2, q\rangle \Downarrow n_2}{\langle e_1 \texttt{ or } e_2, q\rangle \Downarrow n_1 \vee n_2}$$

Statements. In the big-step semantics, we write:

$$\langle s, q\rangle \Downarrow q'$$

which is to say that statement s executed in state q results in state q'. That is, executing a statement transforms a state into a new state.

Here's some notation.

- empty state: []
- an example state: $[x := 10; y := 15; z := 5]$
- substitution: $q[x := 10]$ (yields a state like q but now x is bound to 10.)

OK, so let's give rules for evaluating statements. We say that statements have side-effects because they don't directly return a value in the way that expressions did. Many statements yield a modified heap. print_state doesn't modify the heap but it presumably has an actual side-effect of printing the state somewhere.

$$\frac{}{\langle \mathsf{skip}, q\rangle \Downarrow q} \qquad \frac{}{\langle \mathsf{print\_state}, q\rangle \Downarrow q}$$

The next two rules are statement composition and assignment.

$$\frac{\langle s_1, q\rangle \Downarrow q'' \qquad \langle s_2, q\rangle \Downarrow q'}{\langle s_1 \texttt{ ; } s_2, q\rangle \Downarrow q'} \qquad \frac{\langle e, q\rangle \Downarrow n}{\langle x \texttt{ := } e, q\rangle \Downarrow q[x := n]}$$

We've said that `;` in WHILE is statement composition. Here we compose s_1 and s_2 and the final effect is that of executing them sequentially.

Assignment changes the value of x to whatever e evaluates to.

Next up, we have `if/then/else`.

$$\frac{\langle b, q\rangle \Downarrow \mathsf{true} \qquad \langle s_1, q\rangle \Downarrow q'}{\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, q\rangle \Downarrow q'} \qquad \frac{\langle b, q\rangle \Downarrow \mathsf{false} \qquad \langle s_2, q\rangle \Downarrow q'}{\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, q\rangle \Downarrow q'}$$

Evaluate b. If it turns out to be true, then we run the then-branch s_1. If it's false, then we run s_2.

Example of evaluating statements. Derivation is execution. Let's see how that works. We execute the compound statement `p := 0; x := 1; n := 2` and show that

$$\langle \texttt{p := 0; x := 1; n := 2}, []\rangle \Downarrow [p := 0, x := 1, n := 2]$$

with the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle 0, []\rangle \Downarrow 0}}{\langle p := 0, []\rangle \Downarrow [p := 0]} \quad \dfrac{\dfrac{}{\langle 1, [p := 0]\rangle \Downarrow 1}}{\langle x := 1, [p := 0]\rangle \Downarrow [p := 0, x := 1]}}{\langle p := 0; x := 1, []\rangle \Downarrow [p := 0, x := 1]} \quad \dfrac{\dfrac{}{\langle 2, [p := 0, x := 1]\rangle \Downarrow 2}}{\langle n := 2, [p := 0, x := 1]\rangle \Downarrow [p := 0, x := 1, n := 2]}}{\langle p := 0; x := 1; n := 2, []\rangle \Downarrow [p := 0, x := 1, n := 2]}$$

It's hard to describe in words in this document, but basically, we are building the semantics of the compound statement `p := 0; x := 1; n := 2` by combining the first statement `p := 0` (which yields a state q_1 of $[p := 0]$ when executed on the empty state $[]$) with `x := 1` (yielding state $[p := 0; x := 1]$ when executed on q_1), and then combining `n := 2` with the result of the first two statements composed, to obtain final state $[p := 0, x := 1, n := 2]$.

Semantics of Loops. We give the semantics of the while statement, in two parts. The first part is easy: if the loop condition b evaluates to false, then the while statement is tantamount to skip.

$$\frac{\langle b, q\rangle \Downarrow \mathsf{false}}{\langle \mathsf{while}\ b\ \mathsf{do}\ s, q\rangle \Downarrow q}$$

The other case involves executing the body of the while loop once, and then executing the while statement again, but starting from the state that we got after executing the body once.

$$\frac{\langle b, q\rangle \Downarrow \mathsf{true} \qquad \langle s; \mathsf{while}\ b\ \mathsf{do}\ s, q\rangle \Downarrow q'}{\langle \mathsf{while}\ b\ \mathsf{do}\ s, q\rangle \Downarrow q'}$$

It turns out, though, that not all loops terminate. What about infinite executions? Operating system loops don't terminate, for instance. Neither does execution of reactive systems.

We could introduce state $\top$ (read *top*) to represent divergence, and say that an infinite loop enters that state.

$$\frac{}{\langle \mathsf{while\ true\ do}\ s, q\rangle \Downarrow \top}$$

and just not do anything (like skip) when we are in state $\top$:

$$\frac{}{\langle s, \top \rangle \Downarrow \top}$$

But that's not very interesting. We don't get any insight into what happens. So, instead, we will introduce small-step, or structural, semantics to deal with these kinds of programs—reactive executions, which don't terminate, but produce useful side effects while they are executing.

Digression: properties of semantics. We talked about deterministic versus nondeterministic functions earlier. We now define deterministic semantics. A semantics is *deterministic* if every program statement has at most one possible derivation in any state, i.e.

$$\text{if } \langle s, q \rangle \Downarrow q_1 \text{ and } \langle s, q \rangle \Downarrow q_2 \text{ then } q_1 = q_2.$$

On the other side of the $\Downarrow$, we say that two statements are *semantically equivalent* if we can't distinguish their output states given the same input state:

$$s_1 \text{ and } s_2 \text{ are semantically equivalent if } \forall q.\ \langle s_1, q \rangle \Downarrow q_1 \text{ and } \langle s_2, q \rangle \Downarrow q_2 \text{ implies } q_1 = q_2.$$

Unrolling a while loop yields a semantically equivalent statement: $\mathsf{while}\ b\ \mathsf{do}\ s$ is equivalent to $\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \mathsf{skip}$.

Structural induction. We have another document, L05b, which gives an example of a structural induction. Here's basically how you do it for derivation trees. To prove property P on all possible derivation trees for e.g. a semantics:

- *Base case.* Prove P for all of the axioms.
- Inductive case. We inductively assume that P holds for all smaller derivation trees and show that P continutes to hold when applying each possible rule once.

I can prove that our semantics are deterministic using structural induction. You also should be able to do it.

The concept of structural induction generalizes the induction you've seen on the integers in high school and MATH 135. But there is more than one base case, and more than one successor function. It has to be true, though, that it is always possible to get from any element (of the well-ordered set) to the base case in a finite number of steps.

Small-Step Operational Semantics

To avoid the problem of defining a statement's semantics by the state after its execution completes (and being hosed when the statement doesn't complete), we can instead define *small-step* operational semantics, which says what happens one step at a time. So:

$$\langle s, q \rangle \Rightarrow \langle t, q' \rangle$$

The big changes are that on the left hand side we have the entire program that you are executing, s (rather than just one statement), and on the right hand side of the $\Rightarrow$ relation we have the *rest* of the program t, as well as the output state q'. So we peel off the first statement from s, which we execute, and we have a remaining program to execute. As before, q is the input state, in which we execute s.

We can also give $\Rightarrow$ for the final statement in a program:

$$\langle s, q\rangle \Rightarrow q'$$

where there is no more program left to execute.

The small-step semantics looks similar to the big-step semantics, with subtle changes. In some ways they're simpler.

$$\frac{}{\langle \mathsf{skip}, q\rangle \Rightarrow q}$$

$$\frac{\langle s_1, q\rangle \Rightarrow q'}{\langle s_1\,;s_2, q\rangle \Rightarrow \langle s_2, q'\rangle} \qquad \frac{\langle s_1, q\rangle \Rightarrow \langle s_3, q'\rangle}{\langle s_1\,;s_2, q\rangle \Rightarrow \langle s_3\,;s_2, q'\rangle}$$

$$\frac{\langle b, q\rangle \Downarrow \mathsf{true}}{\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, q\rangle \Rightarrow \langle s_1, q\rangle} \qquad \frac{\langle b, q\rangle \Downarrow \mathsf{true}}{\langle \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2, q\rangle \Rightarrow \langle s_2, q\rangle}$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ s, q\rangle \Rightarrow \langle \mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \mathsf{skip}, q\rangle}$$

Properties of small-step semantics. To be completely confusing, we're going to overload the word "state" here. We don't mean "store". Instead, by state, we mean a configuration $\langle s, q\rangle$ with a remaining-program-to-execute and a store.

You can then view the small-step semantics as a transition system $TS = (S, R)$, where S is a set of states, and R is a transition relation on a pair of states, so that $(x, y) \in R$ iff $x \Rightarrow y$ is a true judgment in the small-step semantics.

With this transition system, we can talk about a path $x_1, x_2, \ldots, x_n$ in TS, and call that a *derivation sequence*, analogous to the derivations we showed for the big-step semantics. Derivation sequences correspond to program executions.

In this case, we can induct (just plain integer induction) on the length of the derivation. Of course, to add one to the length of the derivation, you can take any of the transitions in the semantics.

The small-step semantics is deterministic if there is at most one derivation for every configuration.

Assertions, assumes, and specifications

Code usually isn't considered a specification language. Impementations are too imperative. But we can embed specifications with our code.

As you recall from SE 212, an implementation is correct if it always meets its postconditions when called in a context where the preconditions are satisfied.

The building blocks are asserts, assumes, and the havoc statement. We use assume and havoc to express the precondition, and assert to check the postcondition.

To check an implementation by symbolically executing it, then, we symbolically execute the assumes and havocs that represent the preconditions. This leaves us with a bunch of possible (abstract) executions that satisfy the preconditions. We then go through the body of the implementation. When we reach the asserts that encode the postcondition, then we have to be able to show that all executions that get to these asserts satisfy the postconditions.

Another way of thinking about it is as follows. Upon entry to a method, anything could be true: conceptually, there are tons of possible states at entry. But we want the precondition to be true. So we just *throw away* states where the precondition is not true. No harm, no foul: we haven't tried to do anything yet.

Then we run all of the remaining states through the implementation, and we get to the end, where the postcondition has to hold. We check that the postcondition does indeed hold. If it does, then we know that the code meets the specification. If it doesn't, then the code doesn't meet the specification.

Here's yet another way to say the same thing. A program is correct if all executions that satisfy all assumptions also satisfy all assertions. A program is incorrect if there exists an execution that satisfies all of the assumptions and violates at least one assertion.

I should walk you through a concrete example, but I've spent too much time typing up these notes. Maybe next time I teach this course. Sorry. (It would be fair game to ask you to explain this to me via an example though.)

Well, here's one partial example from the slides.

```
1   int x, y;
2
3   void main(void) { // not standards-compliant C, oh well
4     havoc(x);
5     assume (x > 10);
6     assume (x <= 100);
7
8     y = x + 1;
9
10    assert (y > x);
11    assert (y < 200);
12  }
```

Note that the order matters. The following 2 programs are different and the one on the right, in particular, doesn't really say anything meaningful.

```
1    int x, y;
2
3    void main(void) {
4      havoc(x);
5      y = x + 1;
6
7      assume (x > 10);
8      assume (x <= 100);
9
10     assert (y > x);
11     assert (y < 200);
12   }
```

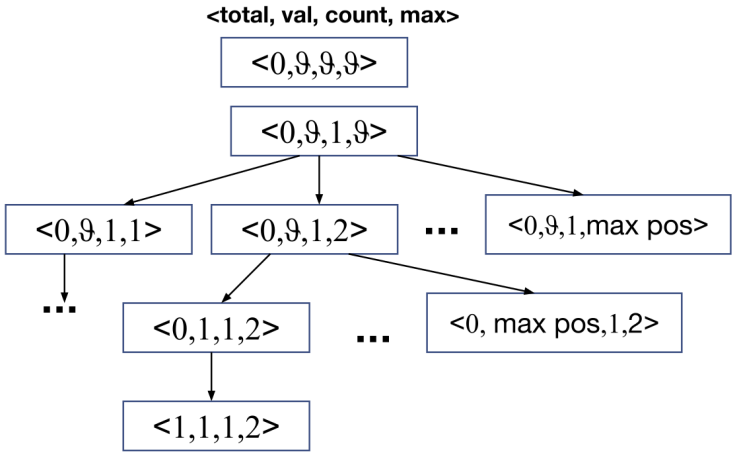
```
1    int x, y;
2
3    void main(void) {
4      havoc(x);
5      y = x + 1;
6
7      assert (y > x);
8      assert (y < 200);
9
10     assume (x > 10);
11     assume (x <= 100);
12   }
```

Graphs as models of computation

We can talk about a computation tree, which is a tree model of all of the possible executions of a system. Each node represents a state of the system (assigns values to all variables); transitions are as in the semantics. These trees can have a infinite number of paths, and a path can be infinite (representing a non-terminating computation).

Here's an example. Is this an infinite tree?

```
1      total := 0;
2      count := 1;
3      max := input();
4      while (count <= max)
5      do {
6        val := input();
7        total := total + val;
8        count := count + 1
9      } ;
10     print (total)
```



The computation tree does indeed represent the space that we want to reason about. But as a thing that you try to work with, it's unwieldy. For anything interesting, it's definitely too large to explicitly create (possibly infinite) or to reason about.

We need abstraction to reason about things. Abstract values, or abstract flow of control. The abstraction you want to use depends on what you're trying to establish.