CS558 Programming Languages

Fall 2015

Lecture 6

INFORMAL SEMANTICS

• Grammars can be used to define the legal programs of a language, but not what they do! (Actually, most languages place further, non-grammatical restrictions on legal programs, e.g., type-correctness.)

• Language behavior is usually described, documented, and implemented on the basis of **natural-language** (e.g., English) descriptions.

• Descriptions are usually structured around the language's grammar, e.g., they describe what each nonterminal does.

• Natural-language descriptions tend to be **imprecise**, **incomplete**, and **inconsistent**.

EXAMPLE: FORTRAN DO-LOOPS

"DO n i = m_1,m_2,m_3

Repeat execution through statement n, beginning with $i = m_1$, incrementing by m_3, while i is less than or equal to m_2. If m_3 is omitted, it is assumed to be 1. m's and i's cannot be subscripted. m's can be either integer numbers or integer variables; i is an integer variable."

- from DEC Fortran-II manual, 1974.

Consider:

```
      DO 100 I = 10,9,1
...
100   CONTINUE
```

How many times is the body executed?

EXPERIMENTAL SEMANTICS

Try it and see!

Implementation becomes standard of correctness.

This is certainly **precise**: compiler source code becomes specification.

But it is:

- difficult to understand;
- awkward to use;
- subject to accidental change;
- wholly non-portable.

FORMAL SEMANTICS

Aims:

• **Rigorous** and **unambiguous** definition in terms of a well-understood formalism, e.g., logic, naive set theory, etc.

• Independence from **implementation**. Definition should describe how the language behaves as abstractly as possible.

Uses:

• Provably-correct implementations.

• Provably-correct programs.

• Basis for language comparison.

• Basis for language design.

(But usually not basis for learning a language.)

Main varieties: Operational, Denotational, Axiomatic.

Each has different purposes and strengths. In this course, we'll mostly focus on operational semantics, with brief looks at the others.

SEMANTICS FROM INTERPRETERS

In the homework, we're building **definitional interpreters** for small languages that display key programming language constructs.

Our goal is to study the interpreter code in order to understand **implementation** issues associated with each language.

In addition, the interpreter serves as a form of **semantic** definition for each language construct. In effect, it defines the meaning of the language in terms of the semantics of Scala.

(Of course, you're also learning more about the semantics of Scala as we go!)

SEMANTICS AND ERRONEOUS PROGRAMS

An important part of a language specification is distinguishing valid from invalid programs.

It is useful to define three classes of errors that make programs invalid. (Of course, even valid programs may behave differently than the programmer intended!)

Static errors are violations of the language specification that can be detected at compilation time (or, in an interpreter, before interpretation begins)

- Includes: **lexical** errors, **syntactic** errors (caught during parsing), **type** errors, etc.

- Compiler or interpreter issues an error pinpointing erroneous location in source program.

- Language **semantics** are usually defined only for programs that have no static errors.

RUNTIME ERRORS

Checked runtime errors are violations that the language implementation is required to detect and report at runtime, in a clean way.

- Examples in Scala or Java: division by zero, array bounds violations, dereferencing a null pointer.
- Depending on language, implementation may issue an error message and die, or raise an exception (which can be caught by the program).
- Language semantics must specify behavior precisely.

Unchecked runtime errors are violations that the implementation need not detect.

- Subsequent behavior of the computation is **arbitrary**. (Error is often not manifested until much later in execution.)
- Examples in C: division by zero, dereferencing a null pointer, array bounds violations.
- Language semantics probably don't specify behavior.
- **Safe** languages like Scala, Java, or Python have **no** such errors!

AXIOMATIC SEMANTICS

Interpreters give an **operational** semantics for imperative statements.

(We'll see other, more formal, operational approaches to semantics shortly.)

An important alternative approach is to give a **logical** interpretation to statements.

- The **state** of an imperative program is defined by the values of the all its variables.

- We can characterize a state by giving a logical **predicate** (or **assertion**), mentioning the variables, which is **satisfied** by the values of the variables in that state.

- We can define the semantics of statements by saying how they affect (arbitrary) predicates.

TRIPLES INVOLVING ASSERTIONS

We write a **Hoare triple**

$$\{\ P\ \}\ \mathtt{S}\ \{\ Q\ \}$$

to mean that if the **precondition** P is true before the execution of S then the **postcondition** Q will be true after the execution of S.

Note that the triple says nothing about what happens if S doesn't terminate. So we are only characterizing statements that terminate.

Examples of triples (not all stating true things!)

$\{y \geq 3\}$ `x := y + 1` $\{x \geq 4\}$

$\{x + y = c\}$ `while x > 0 do`
`y := y + 1;`
`x := x - 1`
`end` $\{x + y = c\}$

$\{\ y = 2\ \}$ `x := y + 1` $\{\ x = 4\ \}$

$\{\ y = 2\ \}$ `x := y + z` $\{\ x = 4\ \}$

Axioms and Rules of Inference

How do we distinguish true triples from false?

Who's to say which ones are true?

It all depends on the **semantics** of statements!

If we work in a suitably structured language, we can give a fixed set of **axioms** and **rules of inference**, one for each kind of statement. We then take as true the set of triples that can be logically **deduced** from these axioms and rules.

Of course, we want to choose axioms and rules that capture our intuitive understanding of what the statements do, and they need to be as **strong** as possible.

ASSIGNMENT AXIOM

$\{ \; P[E/x] \; \}$ `x := E` $\{ \; P \; \}$

where $P[E/x]$ means P with all instances of x replaced by E.

This axiom may seem backwards at first, but it makes sense if we start from the postcondition. For example, if we want to show $x \geq 4$ after the execution of

```
x := y + 1
```

then the necessary precondition is $y + 1 \geq 4$, i.e., $y \geq 3$.

More Rules for Statements

Conditional Rule

$$\frac{\{\ P \wedge E\ \}\ \mathtt{S}_1\ \{\ Q\ \},\ \{\ P \wedge \neg E\ \}\ \mathtt{S}_2\ \{\ Q\ \}}{\{\ P\ \}\ \texttt{if E then } \mathtt{S}_1\ \texttt{else } \mathtt{S}_2\ \texttt{endif}\ \{\ Q\ \}}$$

Composition Rule

$$\frac{\{\ P\ \}\ \mathtt{S}_1\ \{\ Q\ \},\ \{\ Q\ \}\ \mathtt{S}_2\ \{\ R\ \}}{\{\ P\ \}\ \mathtt{S}_1\texttt{; } \mathtt{S}_2\ \{\ R\ \}}$$

While Rule

$$\frac{\{\ P \wedge E\ \}\ \mathtt{S}\ \{\ P\ \}}{\{\ P\ \}\ \texttt{while E do S}\ \{\ P \wedge \neg E\ \}}$$

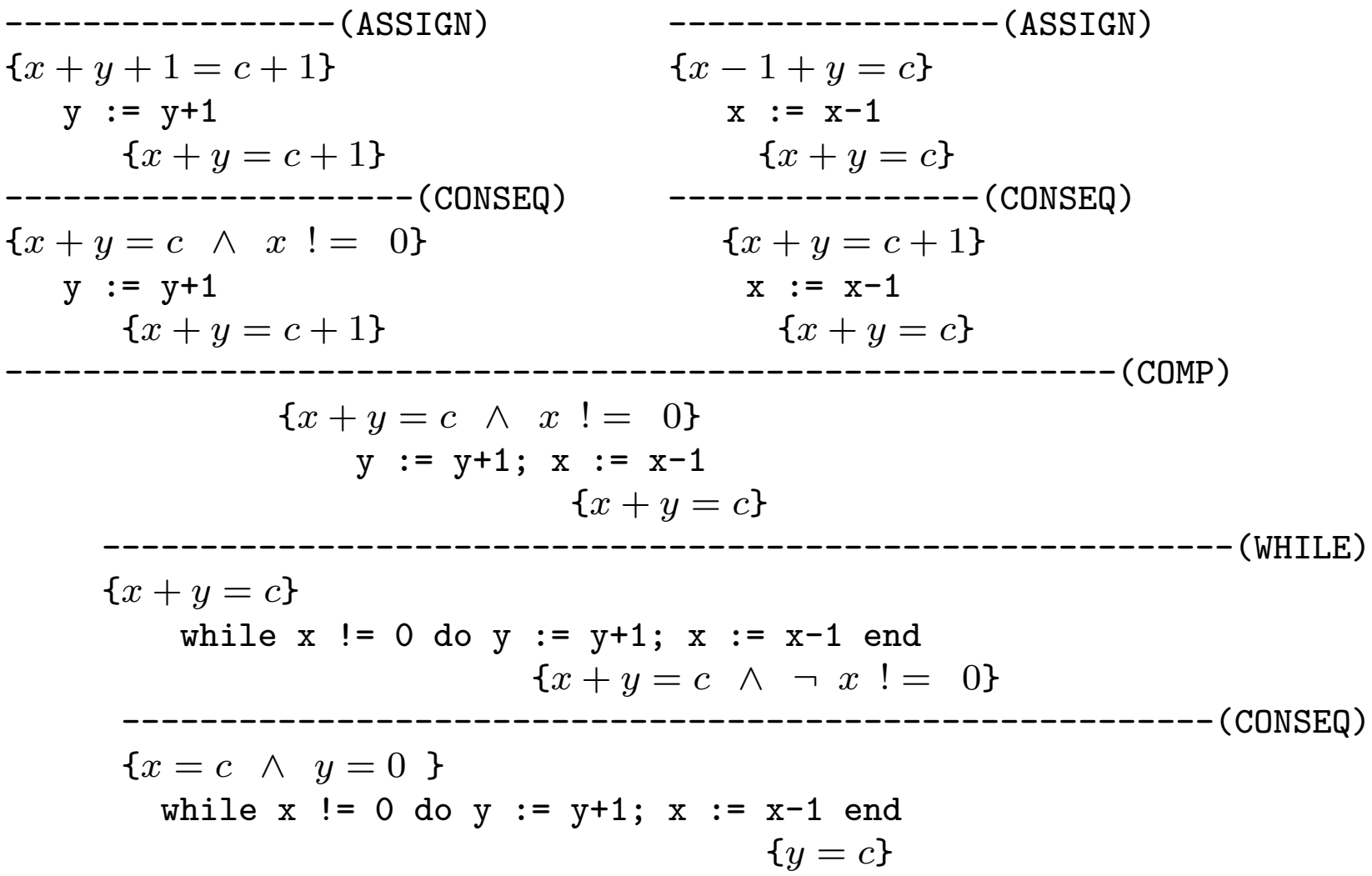
BOOKKEEPING RULES

Consequence Rule

$$\frac{P \;\Rightarrow\; P',\ \{\ P'\ \}\ \mathtt{S}\ \{\ Q'\ \},\ Q' \;\Rightarrow\; Q}{\{\ P\ \}\ \mathtt{S}\ \{\ Q\ \}}$$

Here $P \Rightarrow Q$ means that "P implies Q," i.e., "Q is true whenever P is true," i.e. "P is false or Q is true." Hence we always have *False* $\Rightarrow Q$ for **any** Q !

PROOF TREE EXAMPLE



Annotated Program Example

Proof trees can be unwieldy. Because the structure of the tree corresponds directly to the structure of the program code, it is common to use an alternative representation of proofs in which we annotate programs with assertions/assumptions.

$\{x = c \ \wedge \ y = 0\}$
$\{x + y = c\}$
```
while x != 0 do
```
  $\{x + y = c \ \wedge \ x \ != \ 0\}$
  $\{x + y + 1 = c + 1\}$
```
  y := y + 1;
```
  $\{x + y = c + 1\}$
  $\{x - 1 + y = c\}$
```
  x := x - 1
```
  $\{x + y = c\}$
```
end
```
$\{x + y = c \ \wedge \ \neg \ x \ != \ 0\}$
$\{y = c\}$

To verify that this is a valid proof, we have to check that the annotations are consistent with each other and with the rules and axioms.

Merits and Problems of Axiomatic Semantics

Gives a very clean semantics for structured statements.

But things get more complicated if we add features like:

- expressions with side-effects
- statements that break out of loops
- procedures
- non-trivial data structures and aliases

Useful for formal proofs of program properties (though these are seldom done by hand).

Thinking in terms of assertions is good for **informal** reasoning about programs.

There are beginning to be useful automated theorem proving support tools too.

Other forms of semantic definition also use similar **logical** structures.

OPERATIONAL SEMANTICS

Define behavior of language by a set of mechanical **state transition rules**.

Typical mechanisms:

• Define a simple Von Neumann-style abstract **stack machine** and describe how each syntactic construct can be compiled into stack machine instructions.

• Characterize the state of the abstract machine (environement, store, stack, etc.) and give a set of **evaluation rules** describing how each syntactic construct affects the state.

Some useful things to do with an operational semantics:

• Build an implementation for a real machine by interpreting or compiling the abstract machine code.

• Explicate the meaning of a language feature by proving that it has the same behavior as a combination of simpler features.

• Prove that correctly typed programs have no unchecked runtime errors.

FORMAL OPERATIONAL SEMANTICS

So far, we've presented operational semantics using interpreters. These have the advantage of being **precise** and **executable**. But they are not ideally **compact** or **abstract**.

Another way to present operational semantics is using **state transition judgments**, for appropriately defined machine states.

For example, consider a simple language of imperative expressions, in which variables must be defined before use, using a `let` or `fun` construct.

```
exp := var | int
    |  '(' '+' exp exp ')'
    |  '(' 'let' var exp exp ')'
    |  '(' ':=' var exp ')'
    |  '(' 'ifnz' exp exp exp ')'
    |  '(' 'while' exp exp ')'
    |   etc.
```

For simplicity, we assume that the only values are integers; `ifnz` expects an integer condition and tests whether it is non-zero.

STATE MACHINE

To evaluate this language, we choose a machine state consisting of:

- the current **environment** E, which maps each in-scope variable to a location l.
- the current **store** S, which maps each location l to an integer value v.
- the current **expression** e, to be evaluated.

We give the state transitions in the form of **judgments**:

$$\langle e, E, S \rangle \Downarrow \langle v, S' \rangle$$

Intuitively, this says that evaluating expression e in environment E and store S yields the value v and the (possibly) changed store S'.

Operational Semantics by Inference

To describe the machine's operation, we give **rules of inference** that state when a judgment can be derived from judgments about sub-expressions.

The form of a rule is

$$\frac{\textit{premises}}{\textit{conclusion}}\ \text{(Name of rule)}$$

We can view evaluation of the program as the process of building an inference tree.

This particular formulation, where each expression results in a value based on the values of the subexpessions, is called **big-step semantics**.

This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content of judgments is different.

ENVIRONMENTS AND STORES, FORMALLY

• We write $E(x)$ means the result of looking up x in environment E. (This notation is because an environment is like a **function** taking a name as argument and returning a meaning as result.)

• We write $E + \{x \mapsto l\}$ for the environment obtained from existing environment E by **extending** it with a new binding from x to location l. If E already has a binding for x, this new binding replaces it.

The **domain** of an environment, *dom*(E), is the set of names bound in E.

Analogously with environments, we'll write

• $S(l)$ to mean the value at location l of store S

• $S + \{l \mapsto v\}$ to mean the store obtained from store S by extending (or updating) it so that location l maps to value v.

• *dom*(S) for the set of locations bound in store S.

Evaluation Rules (1)

$$\frac{l = E(x) \quad v = S(l)}{\langle x, E, S\rangle \Downarrow \langle v, S\rangle} \text{ (Var)}$$

$$\frac{}{\langle i, E, S\rangle \Downarrow \langle i, S\rangle} \text{ (Int)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle \texttt{(+ } e_1 \ e_2\texttt{)}, E, S\rangle \Downarrow \langle v_1 + v_2, S''\rangle} \text{ (Add)}$$

$$\frac{\begin{array}{c}\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad l \notin \mathit{dom}(S') \\ \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\}\rangle \Downarrow \langle v_2, S''\rangle\end{array}}{\langle \texttt{(let } x \ e_1 \ e_2\texttt{)}, E, S\rangle \Downarrow \langle v_2, S''\rangle} \text{ (Let)}$$

$$\frac{\langle e, E, S\rangle \Downarrow \langle v, S'\rangle \quad l = E(x)}{\langle \texttt{(:= } x \ e\texttt{)}, E, S\rangle \Downarrow \langle v, S' + \{l \mapsto v\}\rangle} \text{ (Assgn)}$$

EVALUATION RULES (2)

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \quad v_1 \neq 0 \quad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle}{\langle (\texttt{ifnz}\ e_1\ e_2\ e_3), E, S\rangle \Downarrow \langle v_2, S''\rangle}\ \text{(If-nzero)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle \quad \langle e_3, E, S'\rangle \Downarrow \langle v_3, S''\rangle}{\langle (\texttt{ifnz}\ e_1\ e_2\ e_3), E, S\rangle \Downarrow \langle v_3, S''\rangle}\ \text{(If-zero)}$$

$$\frac{\begin{array}{c}\langle e_1, E, S\rangle \Downarrow \langle v_1, S'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, E, S'\rangle \Downarrow \langle v_2, S''\rangle \\ \langle (\texttt{while}\ e_1\ e_2), E, S''\rangle \Downarrow \langle v_3, S'''\rangle\end{array}}{\langle (\texttt{while}\ e_1\ e_2), E, S\rangle \Downarrow \langle v_3, S'''\rangle}\ \text{(While-nzero)}$$

$$\frac{\langle e_1, E, S\rangle \Downarrow \langle 0, S'\rangle}{\langle (\texttt{while}\ e_1\ e_2), E, S\rangle \Downarrow \langle 0, S'\rangle}\ \text{(While-zero)}$$

NOTES ON THE RULES

- The structure of the rules guarantees that at most one rule is applicable at any point.
- The store relationships constrain the order of evaluation.
- If no rules are applicable, the evaluation **gets stuck**; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

```
def eval(e:Exp, env:Env, st:Store) : (Value,Store) = ...
```

returns a value `v` and store $\texttt{st}'$ such that

$$\langle \texttt{e}, \texttt{env}, \texttt{st} \rangle \Downarrow \langle \texttt{v}, \texttt{st}' \rangle$$

The implementation of `eval` dispatches on the syntactic form of `e`, chooses the appropriate rule, and makes recursive calls on `eval` corresponding to the premises of that rule.

Question: how deep can the derivation tree get?