

# CS558 Programming Languages

## Winter 2008

### Lecture 4

#### 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**.

1

PSU CS558 W'08 LECTURE 4 © 1994–2008 ANDREW TOLMACH

2

#### 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.

**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.

Define behavior of language on an **abstract machine**.

Abstract machine should be much **simpler** than real machines, since otherwise a compiler for a real machine would be just as good!

Typical mechanisms:

- Characterize the state of the abstract machine (typically as an **environment** mapping variables to values) and give a set of **evaluation rules** describing how each syntactic construct affects the state.
- Define a simple Von Neumann-style **stack machine** and describe how each syntactic construct can be compiled into stack machine instructions.

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 cannot “dump core” at runtime.

## SEMANTICS FROM INTERPRETERS

In the homework, we'll be 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 Java or ML.

(Of course, you'll also be learning more about the semantics of Java and ML 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.

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

- Examples in Java or ML: 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.
- Java and ML have **no** such errors!

## TRIPLES INVOLVING ASSERTIONS

We write a **Hoare triple**

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

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!)

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

$$\begin{aligned} \{ x + y = c \} & \text{ while } x > 0 \text{ do} \\ & \quad y := y + 1; \\ & \quad x := x - 1 \\ & \text{ end } \{ x + y = c \} \end{aligned}$$

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

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

So far, we've given an **operational** semantics for imperative statements:

- we can translate them into instructions for a simple abstract machine instructions; or
- we can interpret them directly in an existing language (we'll see a more formal treatment of this approach soon).

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.

## 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.,  $y \geq 3$ .

**Conditional Rule**

$$\frac{\{ P \wedge E \} S_1 \{ Q \}, \{ P \wedge \neg E \} S_2 \{ Q \}}{\{ P \} \text{ if } E \text{ then } S_1 \text{ else } S_2 \text{ endif } \{ Q \}}$$

**Composition Rule**

$$\frac{\{ P \} S_1 \{ Q \}, \{ Q \} S_2 \{ R \}}{\{ P \} S_1; S_2 \{ R \}}$$

**While Rule**

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

**BOOKKEEPING RULES**

**Consequence Rule**

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

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**

```

----- (ASSIGN)
{x + y = c}
  y := y+1
  {x + y = c + 1}
----- (CONSEQ) ----- (ASSIGN)
{x + y = c ∧ x > 0}      {x + y = c + 1}
  y := y+1              x := x-1
  {x + y = c + 1}      {x + y = c}
----- (COMP)
      {x + y = c ∧ x > 0}
      y := y+1; x := x-1
      {x + y = c}
----- (WHILE)
{x + y = c}
  while x > 0 do y := y+1; x := x-1 end
      {x + y = c ∧ x ≤ 0}
----- (CONSEQ)
{x + y = c}
  while x > 0 do y := y+1; x := x-1 end
      {x + y = c}

```

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.

```

{x + y = c}
while x > 0 do
  {x + y = c ∧ x > 0}
  {x + y = c}
  y := y + 1;
  {x + y = c + 1}
  x := x - 1
  {x + y = c}
end
{x + y = c ∧ x ≤ 0}
{x + 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).

Thinking in terms of assertions is good for **informal** reasoning about programs. (And there are beginning to be useful automated theorem proving support tools too.)

Other forms of semantic definition, e.g., **natural semantics**, also use similar **logical** structures.