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 = m1,m2,m3
Repeat execution through statement n, beginning with i = m1, incrementing by m3, while i is less than or equal to m2. If m3 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.
```


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 Scheme.
(Of course, you’ll also be learning more about the semantics of Scheme as we go!)

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 } \)

\{ 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

\{ P \wedge E \} S_1 \{ Q \}, \{ P \wedge \neg E \} S_2 \{ Q \}

------------------------------------------
\{ P \} if E then S_1 else S_2 endif \{ Q \}

Composition Rule

\{ P \} S_1 \{ Q \}, \{ Q \} S_2 \{ R \}

---------------------------------------
\{ P \} S_1 ; S_2 \{ R \}

While Rule

\{ P \wedge E \} S \{ P \}

-----------------------------------------------
\{ P \} while E do S \{ P \wedge \neg E \}

Bookkeeping Rules

Consequence Rule

\( P \Rightarrow P' \), \{ 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 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.

\[
\begin{align*}
{x + y = c} \\
\text{while } x > 0 \text{ do} \\
\quad {x + y = c} \land x > 0 \\
\quad {x + y = c} \\
\quad y := y + 1; \\
\quad {x + y = c + 1} \\
\quad x := x - 1 \\
\quad {x + y = c} \\
\text{end} \\
{x + y = c} \land x \leq 0 \\
{x + y = c}
\end{align*}
\]

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.