1. Formal Semantics
Section 1

Formal Semantics
Semantics of Programs

- Syntax defines what programs are *valid*.
- Semantics defines what the valid programs *mean*.

**Informal definition of meaning:**

- Descriptions: technical manuals etc.
  Suffers from impreciseness and ambiguity.
- Implementations: *definitional translators* (e.g. Fortran, C)
  “The program means what *my* compiler says it means”
  Suffers from
  - *Inconsistencies*: bugs in definitional translators
  - *Lack of extensibility and portability*: translators may not be available for all platforms.
Formal Semantics

A precise mathematical definition of the meaning of programming language constructs.

- Validated translators that strictly follow the language definition.
- Applications that can be proven correct with respect to their design.
- Clear, unambiguous meanings for language constructs.
- ... reliable, trustworthy programs and programming systems

Downsides:

- Usually done after a language gains users
- Usually applies only to a part of a language
- Exceptions: CAML, SML etc., and large parts of Java have formally described semantics!
Methods for Defining Formal Semantics

- **Operational Semantics:** Define the behavior of a program in terms of a well-defined, simple machine.
  - Formally describes *how* the program runs.
  - Can be used to synthesize language translators.

- **Denotational Semantics:** Define the meaning of a program in terms of mathematical functions (and their composition)
  - Formally describes *what* the program computes.
  - Can be used to derive reference (prototype) implementations.

- **Axiomatic Semantics:** Define the meaning of a program in terms of how each statement in a language transforms proof obligations (predicates).
  - Formally describes the relationship between what is true at different program points.
  - Can be used to derive program correctness proofs.
Operational Semantics

Usually described as a **Reduction Machine**.

- The *configuration* of a simple reduction machine is given by either:
  - $\langle \text{State} \rangle$ where *State* consists of an *environment* and *store* that assigns values to variables (names), or
  - $\langle E \mid \text{State} \rangle$ where $E$ is the next expression/statement to be evaluated.

Sometimes (e.g., as in the book), the state is treated as the environment alone.

- Operations of the machine are “moves” between configurations, written as $c_1 \Rightarrow c_2$.

Example: $\langle \text{if(true, s}_1, s_2) \mid \text{State} \rangle \Rightarrow s_1 \mid \text{State} \rangle$

- The machine starts from a configuration with an empty environment.
**Operational Semantics**

- Moves of the machine specified using inference rules of the form $\frac{\text{premise}}{\text{consequence}}$ where
  - consequence defines a move that is possible when
  - premise is true

Example:

\[
\langle \text{expr} \mid \text{State} \rangle \Rightarrow \langle \text{expr}' \mid \text{State} \rangle
\]

\[
\langle \text{expr} + \text{expr}'' \mid \text{State} \rangle \Rightarrow \langle \text{expr}' + \text{expr}'' \mid \text{State} \rangle
\]

- Inference rules without premises are called *axioms*.

- Certain moves may be specified by axioms:

Example:

\[
\langle \text{if(true, s}_1, \ s}_2 \mid \text{State} \rangle \Rightarrow \ s}_1 \mid \text{State} \rangle
\]
Executable Specification of Operational Semantics

- An inference rule of the form \( \frac{\text{premise}}{\text{consequence}} \) can be written as a clause in a logic program of the form \( \text{consequence} : \neg \text{premise} \).

- Moves of the reduction machine can thus be written using clauses such as:

  \[
  \text{reduce}(\text{config}(\text{add}(E, E2), \text{State}), \text{config}(\text{add}(E1, E2), \text{State})) :- \\
  \quad \text{reduce}(\text{config}(E, \text{State}), \text{config}(E1, \text{State})).
  \]

  ...  

  \[
  \text{reduce}(\text{config}(\text{if}(\text{true}, S1, S2), \text{State}), \text{config}(S1, \text{State})).
  \]

  ...

- \text{reduce} specifies \textit{one step} in the execution of the reduction machine.

- The execution of the reduction machine is simply a sequence of reduction steps:

  \[
  \text{execute}(S, E) :- \text{reduce}(S, E).
  \]

  \[
  \text{execute}(S, E) :- \text{reduce}(S, \text{NextStep}), \text{execute}(\text{NextStep}, E).
  \]
The meaning of statements and expressions in a program are given in terms of functions.

The functions `eval_expr`, `eval_stmt`, etc that we wrote in earlier classes are called *semantic functions*.

They are executable specifications of denotational semantics.

Definition of semantic functions is usually written using the following notation:

\[ E[e]_{env} = \ldots \]

where `E` is a semantic function, `e` is a piece of syntax to which the definition attaches a meaning, and `env` is an environment.
Recall: the signature of \texttt{eval\_expr}: \texttt{expr \times env \times store \rightarrow value}; “\texttt{expr}”, which is an abstract syntax representation of an expression is a piece of \textit{syntax}. “\texttt{value}” is the meaning attached to that expression.

\texttt{eval\_expr}: \texttt{expr \rightarrow (env \times store \rightarrow value)} is an alternative way of specifying meanings of expressions.

Meaning denoted by values over \textit{semantic domains}.

The notation clearly separates the syntax of programs from their meaning.

Meanings of complex statements made up of different components is defined as the composition of the component’s semantic functions.
Axiomatic Semantics

The meaning of constructs in a program are given in terms of their effect on *assertions* (predicates) about data values at certain program points.

**Example:** If $x > 0$ holds after $x := x + 1$, what should hold *before*?

$x > -1$

Axiomatic semantics for a statement $S$ takes the form

$\{\text{Precondition}\} \ S \ {\text{Postcondition}}$

- **Postcondition:** assertions that hold after $S$ is executed, and
- **Precondition:** assertions that hold before $S$ is executed.

**Examples:**

- $\{x > -1\} x := x + 1 \{x > 0\}$
- $\{x = A, y = B\} t := x; \ x := y; \ y := t \{x = B, y = A\}$
Assertions are boolean conditions on data values.

Assertions are associated with specific program points and state the conditions that always hold whenever that point is reached.

Assertions provide an useful way to do error checking in programs.

**Example:**

\[
\{ \text{stack} ! = \text{NULL} \} \text{pop(stack)} \{ \text{true} \}
\]

In C (using library `assert.h`)

```c
assert(stack != NULL);
pop(stack);
```

C will raise a signal and exit the program whenever the assertion is violated in that execution.
Weakest Precondition

- A weakest precondition $wp(S, Q)$ is the most general assertion that must hold *before* $S$ in order for $Q$ to hold *after* executing statement $S$.

- Used to compute the most general assertion that must hold at the beginning of a program, given an assertion to be satisfied at the program’s end.

**Examples:**

1. $wp(x := e, Q) = Q[e/x]$  
   ( $Q[e/x]$ is an assertion $Q'$ obtained by replacing every $x$ in $Q$ by $e$.)

   $wp(x := x+1, x > 0) = x + 1 > 0 = x > -1$

2. $wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$

3. $wp(if(c, s_1, s_2), Q) = (c = true \rightarrow wp(s_1, Q))$
   \[\wedge (c = false \rightarrow wp(s_2, Q))\]