Semantics

• There is no single widely acceptable notation or formalism for describing semantics

• Operational Semantics
  – Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
Semantics

• To use operational semantics for a high-level language, a virtual machine is needed
• A hardware pure interpreter would be too expensive
• A software pure interpreter also has problems:
  – The detailed characteristics of the particular computer would make actions difficult to understand
  – Such a semantic definition would be machine-dependent
Operational Semantics

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
  - State changes from execution define meaning
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL)
Semantics

- Axiomatic Semantics
  - Based on formal logic (predicate calculus)
  - Original purpose: formal program verification
  - Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
  - The expressions are called assertions
Axiomatic Semantics

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution.
- An assertion following a statement is a *postcondition*.
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition.
- If weakest precondition computable for each language statement, correctness proofs possible.
Axiomatic Semantics

• Pre-post form: \{P\} statement \{Q\}

• An example: \(a = b + 1\) \{a > 1\}
  One possible precondition: \{b > 10\}
  Weakest precondition: \{b > 0\}
Axiomatic Semantics

• **Program proof process**: The postcondition for the whole program is the desired result. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.
Axiomatic Semantics

• An axiom for assignment statements
  \[(x = E) : \{ Q_{x->E} \} x = E \{ Q \} \]

• The Rule of Consequence:

\[ \{P\} S \{Q\}, \; P = P, \; Q = Q \]
\[ \{P\} S \{Q\} \]
Axiomatic Semantics

- An inference rule for sequences

For a sequence $S_1;S_2$:

$\{P_1\} \ S_1 \ \{P_2\}$

$\{P_2\} \ S_2 \ \{P_3\}$

the inference rule is:

$$
\begin{array}{c}
\{P_1\} \ S_1 \ \{P_2\}, \ \{P_2\} \ S_2 \ \{P_3\} \\
\{P_1\} \ S_1; \ S_2 \ \{P_3\}
\end{array}
$$
Axiomatic Semantics

- An inference rule for logical pretest loops

For the loop construct:
{P} while B do S end {Q}
the inference rule is:

\[
\frac{(I \land B) \ S \ {I}}{(I) \ \text{while} \ \ B \ \text{do} \ S \ {I \land \neg B}}
\]

where I is the loop invariant (the inductive hypothesis)
Axiomatic Semantics

- Characteristics of the loop invariant
  I must meet the following conditions:
  - $P \Rightarrow I$ (the loop invariant must be true initially)
  - $\{I\} B \{I\}$ (evaluation of the Boolean must not change the validity of $I$)
  - $\{I \text{ and } B\} S \{I\}$ (I is not changed by executing the body of the loop)
  - $\{I \text{ and } \neg B\} \Rightarrow Q$ (if I is true and B is false, Q is implied)
  - The loop terminates (this can be difficult to prove)
Axiomatic Semantics

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.
Semantics

- Evaluation of axiomatic semantics:
  - Developing axioms or inference rules for all of the statements in a language is difficult
  - It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
Semantics

• Denotational Semantics
  – Based on recursive function theory
  – The most abstract semantics description method
  – Originally developed by Scott and Strachey (1970)
Denotational Semantics

- The process of building a denotational spec for a language (not necessarily easy):
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

- The meaning of language constructs are defined by only the values of the program's variables
Semantics

• The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions
Denotational Semantics

• The state of a program is the values of all its current variables
  \[ s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\} \]

• Let \textsc{Varmap} be a function that, when given a variable name and a state, returns the current value of the variable
  \[ \textsc{Varmap}(i_j, s) = v_j \]
Semantics

• Decimal Numbers
  – The following denotational semantics description maps decimal numbers as strings of symbols into numeric values
Semantics

\[
<\text{dec\_num}> \rightarrow \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\mid <\text{dec\_num}> (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)
\]

\[
M_{\text{dec}}('0') = 0, \quad M_{\text{dec}}('1') = 1, \ldots, \quad M_{\text{dec}}('9') = 9
\]

\[
M_{\text{dec}}(<\text{dec\_num}> '0') = 10 \times M_{\text{dec}}(<\text{dec\_num}>)
\]

\[
M_{\text{dec}}(<\text{dec\_num}> '1') = 10 \times M_{\text{dec}}(<\text{dec\_num}>) + 1
\]

\[
\ldots
\]

\[
M_{\text{dec}}(<\text{dec\_num}> '9') = 10 \times M_{\text{dec}}(<\text{dec\_num}>) + 9
\]
Semantics

• Expressions
  – Map expressions onto $\mathbb{Z} \cup \{\text{error}\}$
  – We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression
Semantics

\[ M_e(<expr>, s) = \]
\[
\text{case } <expr> \text{ of }
\]
\[
<\text{dec\_num}> \rightarrow M_{\text{dec}}(<\text{dec\_num}>, s)
\]
\[
<\text{var}> \rightarrow
\]
\[
\text{if } \text{VARMAP}(<\text{var}>, s) \text{ == } \text{undef}
\]
\[
\text{then } \text{error}
\]
\[
\text{else } \text{VARMAP}(<\text{var}>, s)
\]
\[
<\text{binary\_expr}> \rightarrow
\]
\[
\text{if } (M_e(<\text{binary\_expr}>.<\text{left\_expr}>, s) \text{ == } \text{undef}
\]
\[
\text{OR } M_e(<\text{binary\_expr}>.<\text{right\_expr}>, s) \text{ == } \text{undef})
\]
\[
\text{then } \text{error}
\]
\[
\text{else }
\]
\[
\text{if } (<\text{binary\_expr}>.<\text{operator}> \text{ == } '+') \text{ then }
\]
\[
M_e(<\text{binary\_expr}>.<\text{left\_expr}>, s) +
\]
\[
M_e(<\text{binary\_expr}>.<\text{right\_expr}>, s)
\]
\[
\text{else } M_e(<\text{binary\_expr}>.<\text{left\_expr}>, s) *
\]
\[
M_e(<\text{binary\_expr}>.<\text{right\_expr}>, s)
\]
\[ ...
\]
Semantics

• Assignment Statements
  - Maps a state to a state

\[
M_a(x := E, s) \Delta = \\
\text{if } M_e(E, s) = \text{error} \\
\text{then error} \\
\text{else } s' = \{ <i_1', v_1'>, <i_2', v_2'>, ..., <i_n', v_n'> \}, \\
\text{where for } j = 1, 2, ..., n, \\
v_j' = \text{VARMAP}(i_j, s) \text{ if } i_j <> x \\
= M_e(E, s) \text{ if } i_j == x
\]
Semantics

• Logical Pretest Loops
  – Maps state to state
  – Assume $M_b$ and $M_{sl}$

$$M_l(\text{while } B \text{ do } L, s) \triangleq$$
  if $M_b(B, s) = \text{undef}$
    then error
  else if $M_b(B, s) = \text{false}$
    then $s$
  else if $M_{sl}(L, s) = \text{error}$
    then error
  else $M_l(\text{while } B \text{ do } L, M_{sl}(L, s))$
Semantics

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors.
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions.
- Recursion, when compared to iteration, is easier to describe with mathematical rigor.
Semantics

• Evaluation of denotational semantics:
  – Can be used to prove the correctness of programs
  – Provides a rigorous way to think about programs
  – Can be an aid to language design
  – Has been used in compiler generation systems