Operational Semantics

• Execute the statements of a program on a (real or virtual) machine and compare the machine’s states

• 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 (cont’d)

• 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

• Evaluation of operational semantics:
  – Good if used informally (language manuals, etc.)
  – Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Axiomatic Semantics

• Based on formal logic (**predicate calculus**)

• **Original purpose**: formal program verification

• Axioms or inference rules are defined for each statement type in the language
  – (to allow transformations of logical expressions into more formal logical expressions)

• The logical expressions are called **assertions**
Axiomatic Semantics (cont’d)

• 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.

Axiomatic Semantics Form

• Pre, post form:
   \[ \{P\} \text{ statement } \{Q\} \]
• An example:
  \[ a = b + 1 \quad \{a > 1\} \]
  – One possible precondition:
    \[ \{b > 10\} \quad a = b + 1 \quad \{a > 1\} \]
  – Weakest precondition (WP):
    \[ \{b > 0\} \quad a = b + 1 \quad \{a > 1\} \]

Program Proof Process

• The postcondition for the entire 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 specification, the program is correct.

Axiomatic Semantics: Axioms

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

• The Rule of Consequence:
  \[ \frac{\{P\}S\{Q\}, P \Rightarrow P, Q \Rightarrow Q'}{\{P\}S\{Q'\}} \]
Axiomatic Semantics: Axioms

• An inference rule for sequences of the form $S_1; S_2$
  \[ \{P_1\} S_1 \{P_2\} \]
  \[ \{P_2\} S_2 \{P_3\} \]

\[ \{P_1\} S_1 \{P_2\}, \{P_2\} S_2 \{P_3\} \]

\[ \{P_1\} S_1; S_2 \{P_3\} \]

Axiomatic Semantics: Axioms

• An inference rule for logical pretest loops
  \[ \{P\} \text{while } B \text{ do } S \text{ end } \{Q\} \]

\[ (I \text{ and } B) \ S \{I\} \]
\[ (I) \text{ while } B \text{ do } S \{I \text{ and not } B]\]

where $I$ is the loop invariant (the inductive hypothesis)

Axiomatic Semantics: Axioms

• 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
    • $(I \text{ and not } B) \Rightarrow Q$ -- if $I$ is true and $B$ is false, $Q$ is implied
    • The loop terminates -- can be difficult to prove

Loop Invariant

• 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.

• Exercise: compute the WP for the following loop:
  \[ \text{while } s>1 \text{ do } s=s/2 \text{ end } \{s=1\} \]
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.
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers.

Denotational Semantics

- The process of building a denotational specification for a language:
  - 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

Simple Example: Decimal Numbers

<dec_num> → '0' | '1' | '2' | '3' | '4' | '5' |
       | '6' | '7' | '8' | '9' |
       |<dec_num> ('0' | '1' | '2' | '3' |
       | '4' | '5' | '6' | '7' |
       | '8' | '9')

Map decimal characters onto their decimal equivalent

\[ M_{\text{dec}}('0') = 0, \quad M_{\text{dec}}('1') = 1, \ldots, \quad M_{\text{dec}}('9') = 9 \]
\[ M_{\text{dec}}(<\text{dec_num}> '0') = 10 \times M_{\text{dec}}(<\text{dec_num}>), \]
\[ M_{\text{dec}}(<\text{dec_num}> '1') = 10 \times M_{\text{dec}}(<\text{dec_num}>) + 1 \]
\[ \vdots \]
\[ M_{\text{dec}}(<\text{dec_num}> '9') = 10 \times M_{\text{dec}}(<\text{dec_num}>) + 9 \]

Denotational Semantics: program state

- 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 \text{VARMAP} be a function that, when given a variable name and a state, returns the current value of the variable
\[ \text{VARMAP}(i_j, s) = v_j \]
Example 2: Expressions

- Map expressions onto \( Z \cup \{\text{error}\} \)
- We assume expressions are
  - decimal numbers, variables, or binary expressions
  - having one arithmetic operator and two operands, each of which can be an expression

Expressions (cont’d)

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

Example 3: Assignment Statements

- Maps state sets onto state sets \( U \{\text{error}\} \)

\[
M_a(x := E, s) \Delta = 
\begin{cases} 
\text{if } M_a(E, s) = \text{error} \\
\text{then error} \\
\text{else } s' = \\
\begin{cases} 
<i_1,v'_1>,<i_2,v'_2>,...,<i_n,v'_n> \\
\text{where for } j = 1, 2, ..., n, \\
\text{if } i_j = x \\
\text{then } v'_j = M_a(E, s) \\
\text{else } v'_j = \text{VARMAP}(i_j, s)
\end{cases}
\end{cases}
\]

Logical Pretest Loops

- Maps state sets to state sets \( U \{\text{error}\} \)

\[
M_l(\text{while } B \text{ do } L, s) \Delta = 
\begin{cases} 
\text{if } M_b(B, s) = \text{undef} \\
\text{then error} \\
\text{else if } M_b(B, s) = \text{false} \\
\text{then } s \\
\text{else } s' = M_l(L, s) \\
\text{if } s' = \text{error} \\
\text{then error} \\
\text{else } M_l(\text{while } B \text{ do } L, s')
\end{cases}
\]

IOM 4036. Ch. 3. Syntax & Semantics 85

IOM 4036. Ch. 3. Syntax & Semantics 86

IOM 4036. Ch. 3. Syntax & Semantics 87

IOM 4036. Ch. 3. Syntax & Semantics 88
Loop Meaning

- 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.

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.
- Because of its complexity, it is of little use to language users.

D. Semantics vs O. Semantics

- In operational semantics, the state changes are defined:
  - by coded algorithms.

- In denotational semantics, the state changes are defined:
  - by rigorous mathematical functions.

Summary

- **BNF** and **context-free grammars** are equivalent metalanguages:
  - Well-suited for describing the syntax of programming languages.
- An **attribute grammar** is a descriptive formalism that can describe both the syntax and the semantics of a language.
- Three primary methods of semantics description:
  - **Operational, axiomatic, denotational**