Semantics of Programming Languages

Semantics = “Meaning”
Programming language semantics describe behavior of a language (rather than its syntax).

All Languages have informal semantics:
• e.g., “This expression is evaluated by evaluating the operator expression to obtain the function closure value, and then the argument expressions left-to-right to obtain actual parameter values, and finally executing the function with its formal parameters bound to the actual parameter values.” (fab manual)
• Usually in English; imprecise; assumes implicit knowledge.

Idea of formal semantics:
• Describe behavior in terms of a formalism.
• To be useful, formalism should be simpler and/or better-understood than original language.
• Possible formalisms include logic, mathematical theory, abstract machines

Why bother with formal semantics?
Want a precise description of language behavior that can be used by programmer and implementor.

Formal semantics gives a machine-independent reference for correctness of implementations.

Can be used to prove properties of languages.
• E.g., Security property: a well-typed program cannot “dump core” at runtime.
• May improve language design by encouraging “cleaner” semantics (much as BNF aided language syntax design).

Varieties of Semantics
Traditionally, three rough categories:

Operational Semantics
• Describe behavior in terms of an operational model, such as an abstract machine with a specified instruction set.

Axiomatic Semantics
• Describe behavior using a logical system containing specified axioms and rules of inference.

Denotational Semantics
• Describe behavior by giving each language phrase a meaning (“denotation”) in some mathematical model.

None of these approaches is entirely satisfactory (esp. compared to BNF approach to syntax). No one “best” approach – different forms may be useful for different purposes.
**Syntax and Semantics**

All these kinds of semantics are structured around language syntax. Useful formalisms try to be compositional: the meaning of the whole is based on the meaning of the parts:

- semantics specifies meaning of primitive elements of the language (AST leaves)
- and of combining elements in the language (AST internal nodes)

Semantics can be described or computed by defining an attribute grammar over the language.

**Operational Semantics**

Define behavior of language constructs by describing how they affect the state of an abstract machine.

Abstract machine generally defined by a finite state and a set of legal state transitions (instructions).

- Like a real machine, only simpler.

Semantics is specified by giving a translation from the source language to the instruction set of the abstract machine (a compiler!)

Machine can be high-level (complicated states and instructions) or low-level (simple states and instructions).

- The lower the machine's level, the more is explained by the semantics, but the more complicated they get.
- Note similarity to choice of intermediate code level.

**Simple Abstract Machine**

State =

- Stack of Values
- Global Environment mapping VARs to VALUEs
- Current Instruction Pointer (IP)

Control = List of Instructions:

- ADD, MULT
  - pop top two values from stack, add/multiply them, and push result
- PUSH value
  - push specified value onto stack
- FETCH var
  - fetch value of specified var from environment and push onto stack
- STORE var
  - pop top value from stack and store into specified var
- PRINT
  - pop top value from stack and print it
- HALT

Initially: empty stack & environment; IP at start of list
Use (synthesized) attributes to build list of instructions.

Notation: \([x_1, \ldots, x_n]\) is the list containing elements \(x_1, \ldots, x_n\) and \(x \cdot y\) is the concatenation of lists \(x\) and \(y\).

\[
\begin{align*}
prog &:= \text{stn} \\
\text{stn} &:= \text{stn}_1 \', \; \text{stn}_2 \\
\text{stn}_1 &:= \text{VAR} \; ':=' \; \text{exp} \\
\text{stn}_2 &:= \text{PRINT} \; \text{exp} \\
\text{exp} &:= \text{NUM} \\
\text{exp} &:= \text{VAR} \\
\text{exp} &:= \text{exp}_1 \; '+' \; \text{exp}_2 \\
\text{exp} &:= \text{exp}_1 \; '*' \; \text{exp}_2
\end{align*}
\]

**Example Program**

Program:

\[
\begin{align*}
a &:= 3; \\
a &:= a + 2; \\
\text{PRINT} & \; a
\end{align*}
\]

**Example Translation**

Program:

\[
\begin{align*}
a &:= 3; \\
a &:= a + 2; \\
\text{PRINT} & \; a
\end{align*}
\]

**Sample Execution**

Instructions | Stack | Environment
---|---|---
- | {} |
PUSH 3 | 3 | {} |
STORE a | - | \{a = 3\} |
FETCH a | 3 | \{a = 3\} |
PUSH 2 | 3, 2 | \{a = 3\} |
ADD | 5 | \{a = 3\} |
STORE a | - | \{a = 5\} |
FETCH a | 5 | \{a = 5\} |
PRINT | - | \{a = 5\} prints 5 !! |
HALT | - | \{a = 5\} |
EXAMPLE PROOF USING OPERATIONAL SEMANTICS

Theorem: The stack never underflows

Lemma 1: If the stack has initial size \( n \), then the net effect of executing the instructions corresponding to an expression is to increase the stack size to \( n + 1 \). Moreover, at no point during such execution is the stack size \( < n \).

• Proof: By induction. \( \text{NUM} \) and \( \text{VAR} \) are base cases; \( + \) and \( \ast \) are inductive cases.

Lemma 2: If the stack has initial size \( n \), then the net effect of executing the instructions corresponding to a statement is to leave the stack at size \( n \). Moreover, at no point during such execution is the stack size \( < n \).

• Proof: By induction, with aid of Lemma 1. \( \text{PRINT} \) and \( := \) are the base cases; ‘;’ is the inductive case.

Proof of theorem: Since the program starts with a stack of size 0 and executes a single statement, Lemma 2 proves that the stack never has size \( < 0 \).

AXIOMATIC SEMANTICS

Describe language in terms of assertions about how statements affect predicates on program variables.

The assertion
\[
\{ P \} \ S \ \{ Q \}
\]
says that if \( P \) is true before the execution of \( S \), then \( Q \) will be true after the execution of \( S \).

Examples:
\[
\{ y \geq 3 \} \ x := y + 1 \ \{ x \geq 4 \}
\]
\[
\{ y = 0 \land x = c \}
\]
\[
\text{while } x > 0 \text{ do}
\]
\[
y := y + 1;
\]
\[
x := x - 1
\]
\[
\text{end}
\]
\[
\{ x = 0 \land y = c \}
\]

USES OF AXIOMATIC SEMANTICS

May be used for proving that a program implements a specification

• i.e. given axioms and rules of inference of the language, show that a given assertion about a given program is true.

Example: Prove
\[
\{ k > 0 \} \text{Prog} \ \{ \text{sum} = \sum_{n=1}^{k} n \}
\]

• where Prog is
\[
i := k; \text{sum} := k;
\]
\[
\text{while } i > 1 \text{ do}
\]
\[
i := i - 1;
\]
\[
\text{sum} := \text{sum} + i
\]
\[
\text{end};
\]

• Can be done by repeated application of axioms and rules. Axiomatic methods become somewhat unwieldy in presence of side-effects and aliasing (multiple names for one storage location). For handling real programs, automated ‘proof assistant’ is essential.
**Example Proof of Correctness in Annotation Form**

\[
\begin{align*}
\{ k > 0 \} \\
\{ k = \sum_{n=k}^{k} n \land k > 0 \} \\
i := k; \\
\{ k = \sum_{n=i}^{k} n \land i > 0 \} \\
\text{sum} := k; \\
\{ \text{sum} = \sum_{n=i}^{k} n \land i > 0 \} \\
\text{while } i > 1 \text{ do} \\
\{ \text{sum} = \sum_{n=i}^{k} n \land i > 0 \land i > 1 \} \\
i := i - 1; \\
\{ \text{sum} = \sum_{n=i+1}^{k} n \land i > 0 \} \\
\text{sum} := \text{sum} + i; \\
\{ \text{sum} = \sum_{n=i}^{k} n \land i > 0 \} \\
\text{end;} \\
\{ \text{sum} = \sum_{n=i}^{k} n \land i > 0 \land i > 1 \} \\
\{ \text{sum} = \sum_{n=i}^{k} n \} 
\end{align*}
\]

**Denotational Semantics**

Program statements and expressions denote mathematical functions between abstract semantic domains.

- In particular, the program as a whole denotes a function from some domain of inputs to some domain of answers.

Semantics are specified as a set of denotation functions mapping pieces of program syntax to suitable mathematical functions.

- Functions are attached to corresponding grammatical constructs using synthesized attribute grammars.

Proper definition of semantic domains is complicated subject – we’ll ignore.

Common notation: \( \lambda x. e \) is an anonymous function with argument \( x \) and body \( e \).

\( \lambda x. x+1 \) \( \lambda y. \text{if } y < 0 \text{ then } -y \text{ else } y \)

**Semantic Domains:**

\[
\begin{align*}
V &= \text{Int (values)} \\
\text{Ide} &= \text{identifiers} \\
S &= \text{Ide} \rightarrow V \quad \text{(stores)} \\
\text{Exp} &= S \rightarrow V \quad \text{(expressions)} \\
\text{Stm} &= S \rightarrow S \quad \text{(statements)}
\end{align*}
\]

**Denotation Functions (from syntactic class to semantic domain):**

\[
\begin{align*}
\text{stm} \rightarrow \text{ID} &: \text{exp} \\
S[\text{stm}] &= \lambda s. \text{update}(s, \text{I[ID]}, E[\text{exp}]s) \\
\text{stm} \rightarrow \text{stm}_1; \text{stm}_2 \\
S[\text{stm}] &= \lambda s. S[\text{stm}_2](S[\text{stm}_1]s) \\
\text{exp} \rightarrow \text{NUM} \\
E[\text{exp}] &= \lambda s. \text{N[NUM]} \\
\text{exp} \rightarrow \text{ID} \\
E[\text{exp}] &= \lambda s. s(\text{I[ID]}) \\
\text{exp} \rightarrow \text{exp}_1 + \text{exp}_2 \\
E[\text{exp}] &= \lambda s. \text{plus}(E[\text{exp}_1]s, E[\text{exp}_2]s) \\
\text{exp} \rightarrow (\text{exp}_1) \\
E[\text{exp}] &= E[\text{exp}_1] \\
\text{N[NUM]} &= \text{NUM}.\text{num} \\
\text{I[ID]} &= \text{ID}.\text{ident}
\end{align*}
\]

**Auxiliary functions:**

\[
\begin{align*}
\text{plus} &: V \times V \rightarrow V \\
\text{update} &: (S \times \text{Ide} \times V) \rightarrow S
\end{align*}
\]
Definition of update:

\[ \text{update} = \lambda (s, id, v). \]
\[ \lambda id_1. \text{if } id = id_1 \text{ then } v \text{ else } s id_1 \]

Fact A: For any \( s_0, x, i \):

\[ (\text{update}(s_0, x, i)) x = (\lambda id_1. \text{if } x = id_1 \text{ then } i \text{ else } s_0 x) x = (\text{if } x = x \text{ then } i \text{ else } s_0 x) = i \]

Fact B: For any \( s_0, x, i, j \):

\[ \text{update}(\text{update}(s_0, x, i), x, j) = \text{update}(s_0, x, j) \]

\[ \lambda v. (\lambda u. \text{update}(u, a, \text{plus}(u(a), 2)))((\lambda t. \text{update}(t, a, 3))(v)) = \lambda v. (\lambda u. \text{update}(u, a, \text{plus}(u(a), 2)))((\lambda t. \text{update}(t, a, 3))(v)) = \lambda v. \text{update}(\text{update}(v, a, 3), a, \text{plus}((\text{update}(v, a, 3))(a), 2)) = \lambda v. \text{update}(\text{update}(v, a, 3), a, \text{plus}(3, 2)) \quad \text{ (using Fact A)}
\]
\[ = \lambda v. \text{update}(\text{update}(v, a, 3), a, 5) = \lambda v. \text{update}(v, a, 5) \quad \text{ (using Fact B)} \]