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

**Source language AST Grammar**

- Designed for easy readability

```
prog ::= stm
stm ::= stm1 ';' stm2
stm ::= VAR ' := ' exp
stm ::= PRINT exp
exp ::= NUM
exp ::= VAR
exp ::= exp1 '+' exp2
exp ::= exp1 '*' exp2
```

- Ambiguity of the grammar doesn't matter, since it's for ASTs.

Very simplistic: no control flow, procedures, datatypes, etc.
SYNTAX-DIRECTED SEMANTICS DEFINITION

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 \(x \odot y\) is the concatenation of lists \(x\) and \(y\).

\[
\begin{align*}
\text{prog} & := \text{stm} & \text{prog.p} & := \text{stm.p} @ [\text{HALT}] \\
\text{stm} & := \text{stm}_1 \; \text{';} \; \text{stm}_2 & \text{stm.p} & := \text{stm}_1.p @ \text{stm}_2.p \\
\text{stm} & := \text{VAR} \; \text{'='} \text{exp} & \text{stm.p} & := \text{exp.p} @ [\text{STORE VAR.var}] \\
\text{exp} & := \text{NUM} & \text{exp.p} & := [\text{PUSH NUM.num}] \\
\text{exp} & := \text{VAR} & \text{exp.p} & := [\text{FETCH VAR.var}] \\
\text{exp} & := \text{exp}_1 \; \text{'}+\text{'} \; \text{exp}_2 & \text{exp.p} & := \text{exp}_1.p @ \text{exp}_2.p @ [\text{ADD}] \\
\text{exp} & := \text{exp}_1 \; \text{'}*\text{'} \; \text{exp}_2 & \text{exp.p} & := \text{exp}_1.p @ \text{exp}_2.p @ [\text{MULT}]
\end{align*}
\]

Example translation

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

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

Example Program

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 | - | \(\{a = 5\}\)
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 \( * \) 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 \( \text{:=} \) 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:

\[
\{ y \geq 3 \} x := y + 1 \{ x \geq 4 \}
\]
\[
\{ y = 0 \land x = c \}
\]

\begin{align*}
&\text{while } x > 0 \text{ do} \\
&\quad y := y + 1; \\
&\quad x := x - 1 \\
&\text{end}
\end{align*}

\[
\{ x = 0 \land y = c \}
\]

Uses of Axiomatic Semantics

May be used for proving programs "correct"

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

Example: Prove

\[
\{ k > 0 \} \text{Prog} \{ \sum_{n=1}^{k} n \}
\]

- where \( \text{Prog} \) is

\begin{align*}
&i := k; \quad \text{sum} := k; \\
&\text{while } i > 1 \text{ do} \\
&\quad i := i - 1; \\
&\quad \text{sum} := \text{sum} + i \\
&\text{end}
\end{align*}

- 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

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

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 \( e \).

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

DENOTATIONAL SEMANTICS OF STRAIGHT-LINE PROGRAMS

Semantic domains:

- \( V = \text{Int} \) (values)
- \( \text{Ide} \) (identifiers)
- \( S = \text{Ide} \to V \) (stores)
- \( \text{Exp} = S \to V \) (expressions)
- \( \text{Stm} = S \to S \) (statements)

Denotation functions (from syntactic class to semantic domain):

\( I : \text{ID} \to \text{Ide} \)
\( N : \text{NUM} \to V \)
\( E : \text{exp} \to \text{Exp} \)
\( S : \text{stm} \to \text{Stm} \)

Auxiliary functions:

- \( \text{plus} : V \times V \to V \)
- \( \text{update} : (S \times \text{Ide} \times V) \to S \)

DENOTATION FUNCTIONS

\( \text{stm} \to \text{ID} := \text{exp} \)
\( \text{S} = \lambda s. \text{update}(s, \text{I[ID]}, \text{E[exp]}s) \)
\( \text{stm} \to \text{stm}1;\text{stm}2 \)
\( \text{S} = \lambda s. \text{S[stm}2](\text{S[stm}1]s) \)
\( \text{exp} \to \text{NUM} \)
\( \text{E[exp]} = \lambda s. \text{N[NUM]} \)
\( \text{exp} \to \text{ID} \)
\( \text{E[exp]} = \lambda s. s(\text{I[ID]})) \)
\( \text{exp} \to \text{exp}1 + \text{exp}2 \)
\( \text{E[exp]} = \lambda s. \text{plus}(\text{E[exp]}s, \text{E[exp]}s) \)
\( \text{exp} \to (\text{exp}1) \)
\( \text{E[exp]} = \text{E[exp]}1 \)

\( \text{N[NUM]} = \text{NUM.num} \)
\( \text{I[ID]} = \text{ID.ident} \)
**Facts About Stores and Updates**

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

**Calculating the Denotation of a Program**

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