A programming language possesses two fundamental features: syntax and semantics. Syntax refers to the appearance of the well-formed programs of the language, and semantics refers to the meanings of these programs. A language’s syntax can be formalized by a grammar or syntax chart; such a formalization is found in the back of almost every language manual. A language’s semantics should be formalized as well, so that it can appear in the language manual, too.

Semantics definition methods fall roughly into three groups:

—**Operational**: The meaning of a well-formed program is the trace of computation steps that results from processing the program’s input.

—**Denotational**: The meaning of a well-formed program is a mathematical function from input data to output data. The steps taken to calculate the output are unimportant; it is the relation of input to output that matters.

—**Axiomatic**: A meaning of a well-formed program is a logical proposition (a “specification”) that states some property about the input and output.

### A Survey of Semantics Methods

We briefly survey the semantic methods by applying them to the oldest and simplest programming language, arithmetic. The syntax of our arithmetic language is:

$$E ::= N \mid E_1 + E_2$$

where $N$ stands for the set of numerals {0, 1, 2, . . .}.

**Operational Semantics.** The simplest version of operational semantics for arithmetic is called a *term rewriting system*; it uses rewriting rule schemes to generate computation steps. There is just one rule scheme for arithmetic; it states that adjacent numerals can be added together:

$$N_1 + N_2 \Rightarrow N',$$

where $N'$ is the sum of the numerals $N_1$ and $N_2$. An operational semantics of a program is the sequence of computation steps generated by the rewriting rule scheme. For example, an operational semantics of the program $(1 + 2) + (4 + 5)$ goes as follows:

$$(1 + 2) + (4 + 5) \Rightarrow 3 + (4 + 5) \Rightarrow 3 + 9 \Rightarrow 12$$

A *structural operational semantics* is a term rewriting system plus a set of inference rule schemes that state the exact context in which a computation step can be undertaken [Hennessy 1991]. A left-to-right computation of arithmetic expressions can be encoded with the previous rewriting rule scheme plus two additional inference rule schemes that (i) always allow a leftmost addition and (ii) allow a rightmost addition only if the subexpressions to the left have evaluated to numerals:

$$E_1 \Rightarrow E'_1 \quad \frac{E_1 + E_2 \Rightarrow E'_1 + E_2}{N + E_2 \Rightarrow N + E'_2}$$

These rules allow the above rewriting of $(1 + 2) + (4 + 5)$ to 12 to stand, but a
right-to-left addition, like \((1 + 2) + (4 + 5) \Rightarrow (1 + 2) + 9\), is impossible to derive.

**Denotational Semantics.** Denotational semantics emphasizes that a program has an underlying mathematical meaning that is structured on the language's syntax definition [Schmidt 1986; Winskel 1993]. For arithmetic, a function, \(e\), maps arithmetic expressions to their meanings in \(\text{Nat} = \{0, 1, \ldots\}\); the meaning of a numeral is just itself, and the meaning of an addition is, of course, the sum of the meanings of the addition's components. We write this as follows:

\[
e : \text{Expression} \rightarrow \text{Nat}
\]

\[
e[N] = N
\]

\[
e[E_1 + E_2] = \text{plus}(e[E_1], e[E_2])
\]

For clarity, square double brackets enclose pieces of syntax; note that \(\text{plus} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat}\) maps two numbers to their sum. Here is the semantics of the example program:

\[
e[1 + 2] + (4 + 5)] = \text{plus}(e[1 + 2], e[4 + 5])
\]

\[
= \text{plus}(\text{plus}(e[1], e[2]), \text{plus}(e[4], e[5]))
\]

\[
= \text{plus}(\text{plus}(1, 2), \text{plus}(4, 5))
\]

\[
= \text{plus}(3, 9) = 12
\]

**Axiomatic Semantics.** An axiomatic semantics uses a set of logic rules to derive properties of programs rather than meanings. As an example in arithmetic, say that we wish to derive even-odd properties, \(\{\text{is\_even}, \text{is\_odd}\}\). We can define an axiomatic semantics to do this:

\[
\begin{align*}
N &: \text{is\_even if } N \mod 2 = 0 \\
N &: \text{is\_odd if } N \mod 2 = 1
\end{align*}
\]

\[
E_1 : p_1 \quad E_2 : p_2 \\
E_1 + E_2 : p_3
\]

where \(p_3 = \begin{cases} \text{is\_even if } p_1 = p_2 \\
\text{is\_odd otherwise} \end{cases}\)

The derivation of the even-odd property of our example program is:

\[
\begin{align*}
1 &: \text{is\_odd} \\
2 &: \text{is\_even} \\
4 &: \text{is\_even} \\
5 &: \text{is\_odd} \\
(1 + 2) + (4 + 5) &: \text{is\_even}
\end{align*}
\]

For imperative programming languages, program properties are stated in the form of Hoare triples; an example is \(X \geq 1: Y := X + 2 \langle Y > 3 \rangle\), which asserts, if variable \(X\) is greater than 1 prior to the execution of \(Y := X + 2\), then \(Y > 3\) holds afterwards [Nielson and Nielson 1992].

**APPLICATIONS OF SEMANTICS**

A pragmatist might view an operational or denotational semantics as merely an "interpreter" for a programming language. Thus, to define a semantics for a general-purpose programming language, one writes an interpreter that manipulates data structures like symbol tables (environments) and storage vectors (stores). For example, a denotational semantics for an imperative language might use an environment, \(e\), and a store, \(s\), along with an environment lookup operation, \(\text{find}\), and a storage update operation, \(\text{update}\):

\[
\text{C:Command} \\
\rightarrow ((\text{Environment} \times \text{Store}) \rightarrow \text{Store})
\]

\[
\text{C}[I : = E][e, s] = \text{update}(\text{find}(I, e), e[E][e, s], s)
\]

\[
\text{C}[C_1; C_2](e, s) = \text{C}[C_2](\text{C}[C_1](e, s))
\]

\(e\) is the function that gives meanings to expressions. The equations explain that an assignment creates a new storage

ACM Computing Surveys, Vol. 28, No. 1, March 1996
vector from the existing one, and a command composition evaluates its two component commands in turn: the store produced by the first command is passed to the second.

Since data structures like symbol tables and storage vectors are explicit, a language's subtleties are stated clearly and its flaws are exposed as awkward codings in the semantics. This helps a designer tune the language's definition and write a better language manual. With a semantics definition in hand, a compiler writer can produce a correct implementation of the language. Similarly, a user can study the semantics definition instead of writing random test programs.

The semantics methods surveyed here have been applied successfully to define and refine a variety of general-purpose programming languages: Ada, Scheme, and Standard ML are a few recent examples. Probably the most significant application of semantics definitions has been to rapid prototyping—a prototyping tool, like SIS, MESS, Actress, or Typol, inputs a language's semantic definition and translates it into a correct compiler for the language [Lee 1989].

RESEARCH ISSUES IN SEMANTICS

New language paradigms present new challenges to semantic methods. In the functional programming paradigm, a higher-order functional language can use functions as arguments to other functions. In denotational semantics, domain theory [Gunter 1992; Schmidt 1986] is used to formalize such situations with algebraic equations. For example, the set of Values for a Scheme-like language takes the form 
\[ \text{Value} = \text{Nat} + (\text{Value} \to \text{Value}) \], that is, values are numbers or functions on values. Of course, Cantor proved it is impossible to define Value to contain all total functions, so domain theory uses the concept of continuous function from topology to define the Value set so that the functions it contains are just the continuous ones.

Challenging issues also arise in the object-oriented programming paradigm, where objects can be parameters (“messages”) to other objects’ procedures (“methods”), and coercion laws based on inheritance allow controlled mismatches between actual and formal parameters. For example, a procedure that expects an actual-parameter object of type semigroup will accept one of type group, because groups possess the operations of semigroups (plus unit and inverse). Indeed, it is typical to define the group type as an extension of semigroup; we say that a group inherits the structure of a semigroup and therefore groups are coercible to semigroups. Surprisingly, such naive coercions can produce type-unsafe programs, so denotational semantics has been used to formalize safe coercions for inheritance [Gunter and Mitchell 1994].

Yet another challenging topic is parallelism and communication as they arise in the distributed programming paradigm. Here, multiple processes run in parallel and synchronize through communication. Structural operational semantics has been adapted to formalize systems of processes and to study the varieties of communication the processes might undertake [Milner 1989].

REFERENCES


