CSc 520

Principles of Programming Languages

50: Semantics — Introduction

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
In order for
1. compiler writers to know exactly how to implement a language, and
2. language users to know exactly what (combinations of) language constructs mean,
the meaning of a language needs to be defined.

Most definitions of real languages are in a stylized, but informal, English.

It is also possible to give formal semantic language definitions.
In practice, most languages are not defined in a formal, precise, mathematical way.

There have been some attempts, for example Modula-2, Algol 68, and PL/I.

“Simple” languages such as Scheme and Haskell are comparatively easy to define formally, compared to C, C++, Java, etc.
The Modula-2 specification was written in *VDM-SL* (Vienna Development Method - Specification Language), a formalism for giving a precise definition of a programming language in a denotational style.

It was over 500 pages long, and didn’t include specifications of the standard libraries.

Wirth’s original Modula-2 report was 28 pages.

For a history of this disastrous standardization effort, see http://www.scifac.ru.ac.za/cspt/sc22wg13.htm.

Note also that Modula-2 is a very simple language compared to Ada, C++, Java, etc.
VDL (Vienna Definition Language) was used to specify PL/I.

A specification has two parts:
1. A **translator** that specified a translation into an abstract syntax tree,
2. an **interpreter** of the abstract syntax tree.

VDL is a kind of **operational semantics**.

PL/I is large and complex.

The resulting (large) document was called the **Vienna Telephone Directory**. It was impossible to comprehend.
In this class we will consider two methods for defining the semantics of programming languages:

- **Operational semantics** define a computation by giving step-by-step transformations on an abstract machine that simulate the execution of the program.

- **Denotational semantics** constructs a mathematical object (typically a function) which is the meaning of the program.
A compiler performs syntactic and semantic analysis. There really isn’t a sharp distinction between the two.

```java
Is String x; …; print x/2 a syntactic or semantic error?
```

Some would say that it violates the **static semantic rules** of the language, and hence is a semantic (not a syntactic) error.

Others would say it violates **context-sensitive syntax rules** of the language. I.e., they’d consider the program as a whole to determine if it is **well-formed** or not.

We will use the term **contextual constraints** for those rules that restricts the programs which are considered well-formed.
Operational Semantics specifies a language through the steps by which each program is executed.

This is often done informally. For example, the statement $\texttt{while } E \texttt{ do } C$ is specified as

1. Evaluate $E$ to a truthvalue $B$;
2. If $B = \text{true}$ then execute $C$, then repeat from 1).
3. If $B = \text{false}$, terminate.

The emphasis is on specifying the steps needed to execute the program. This makes the specification useful for language implementers.
Operational Semantics...

We need two things:
1. an abstract syntax, and
2. an interpreter.

The abstract syntax defines the structure of each construct in the language, for example, that an if-statement consists of three parts: the test $e$, the then-part $c_1$ and the else-part $c_2$:

$$\text{if} \ ::= \ e \text{:bool_expr} \ c_1 \text{:statement} \ c_2 \text{:statement}$$

Note that no syntactic information is given.

The interpreter generates a sequence of machine configurations that define the program’s semantics. The interpreter is defined by rewriting rules.
Operational Sem. — Peano Arithmetic

Abstract Syntax ($N \in \text{Nat}$, the Natural Numbers):

$$N ::= 0 | S(N) | (N + N) | (N \times N)$$

Interpreter:

$$I : N \rightarrow N$$

$$I \llbracket (n + 0) \rrbracket \Rightarrow n$$
$$I \llbracket (m + S(n)) \rrbracket \Rightarrow S(I \llbracket (m + n) \rrbracket)$$
$$I \llbracket (n \times 0) \rrbracket \Rightarrow 0$$
$$I \llbracket (m \times S(n)) \rrbracket \Rightarrow I \llbracket ((m \times n) + m) \rrbracket$$

where $m, n \in \text{Nat}$
The rewrite rules are used to turn an expression into standard form, containing only $S(\text{succ})$ and 0.

$S(S(S(S(0)))) = 4$. 

[11]
Operational Sem. — Simple...

- *Simple* is a language with if-statements, while-statements, assignment-statements, and integer arithmetic.
- The semantic function $I$ interprets commands.
- The semantic function $\nu$ interprets expressions.
- The store $\sigma$ maps variables to their values.
- Assignments update the store.
- The result of the interpretation (the semantics of the program) is the resulting store.
Operational Sem. — Simple...

Interpreter:

\[ I : C \times \Sigma \rightarrow \Sigma \]
\[ \nu : E \times \Sigma \rightarrow T \cup Z \]

Semantic Equations:

\[ I(\text{skip}, \sigma) = \sigma \]
\[ I(V := E, \sigma) = \sigma[V \mapsto \nu(E, \sigma)] \]
\[ I(C_1 ; C_2, \sigma) = E(C_2, E(C_1, \sigma)) \]
\[ I(\text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end}, \sigma) = \]
\[ I(C_1, \sigma) \text{ if } \nu(E, \sigma) = \text{true} \]
\[ I(C_2, \sigma) \text{ if } \nu(E, \sigma) = \text{false} \]
Operational Sem. — Simple...  

Interpreter:

\[
\text{while } E \text{ do } C \text{ end } = \begin{cases} 
\text{if } E \text{ then } (C; \text{ while } E \text{ do } C \text{ end}) \text{ else skip} \\
\nu(V, \sigma) = \sigma[V] \\
\nu(N, \sigma) = N \\
\nu(E_1 + E_2, \sigma) = \nu(E_1, \sigma) + \nu(E_2, \sigma) \\
\nu(E_1 = E_2, \sigma) = \begin{cases} 
\text{true} \text{ if } \nu(E, \sigma) = \nu(E, \sigma) \\
\text{false} \text{ if } \nu(E, \sigma) \neq \nu(E, \sigma) 
\end{cases}
\end{cases}
\]
We think of each program as implementing a mathematical function.

An imperative program is a function from inputs to outputs. This function is the meaning of the program.

Example

\[
\text{exec } \text{[while } E \text{ do } C]\] = \\
\text{let } \text{exec-while } \text{env } \text{sto} = \\
\text{let } \text{Boolean } \text{tr} = \text{evaluate } [E] \text{ env } \text{sto } \text{in} \\
\text{if } \text{tr } \text{then} \\
\text{exec-while } \text{env } (\text{exec } [C] \text{ env } \text{sto}) \\
\text{else } \text{sto} \\
\text{in} \\
\text{exec-while}
\]
We need three things:

1. an abstract syntax,
2. a semantic algebra defining a computational model, and
3. valuation functions.

The valuation functions map the syntactic constructs of the language to the semantic algebra.

Denotational semantics relies on defining an object in terms of its constituent parts.
Abstract Syntax ($N \in \text{Nat}$, the Natural Numbers):

\[ N ::= 0 \mid S(N) \mid (N + N) \mid (N \times N) \]

Semantic Algebra:

\[ + : \text{Nat} \to \text{Nat} \to \text{Nat} \]

Valuation Function:

\[ D : \text{Nat} \to \text{Nat} \]

\[
\begin{align*}
D[(n + 0)] &= D[n] \\
D[(m + S(n))] &= D[(m + n)] + 1 \\
D[(n \times 0)] &= 0 \\
D[(m \times S(n))] &= D[((m \times n) + m)]
\end{align*}
\]

where $m, n \in \text{Nat}$
Denotational Sem. — Simple

Abstract Syntax:

- **C ∈ Command**
- **E ∈ Expression**
- **O ∈ Operator**
- **N ∈ Numeral**
- **V ∈ Variable**

\[
C ::= V := E \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end} \mid \text{while } E \text{ do } C \text{ end} \mid C_1 \; ; \; C_2 \mid \text{skip}
\]

\[
E ::= V \mid N \mid E_1 \; O \; E_2 \mid (E)
\]

\[
O ::= + \mid - \mid * \mid / \mid = \mid < \mid > \mid <>
\]
Denotational Sem. — Simple...

Semantic Algebra:

\[ \tau \in T = true, false; \text{ the boolean values} \]
\[ \zeta \in Z = \{\ldots -1, 0, 1, \ldots\}; \text{ the integers} \]
\[ + : Z \rightarrow Z \rightarrow Z \]
\[ = : Z \rightarrow Z \rightarrow T \]
\[ \sigma \in S = \text{Variable} \rightarrow \text{Numeral}; \text{ the state} \]

Valuation Functions:

\[ C \in C \rightarrow (S \rightarrow S) \]
\[ E \in E \rightarrow E \rightarrow (N \cup T) \]
\[
C \text{[skip]} \sigma = \sigma
\]
\[
C \text{[} V := E \text{]} \sigma = \sigma [ V \mapsto E \text{[..]} ] \sigma
\]
\[
C \text{[} C_1 ; C_2 \text{]} = C \text{[} C_2 \text{]} C \text{[} C_1 \text{]}
\]
\[
C \text{[if } E \text{ then } C_1 \text{ else } C_2 \text{ end}] \sigma = C \text{[} C_1 \text{]} \sigma \text{ if } E \text{[} E \text{]} \sigma = \text{true}
\]
\[
C \text{[if } E \text{ then } C_1 \text{ else } C_2 \text{ end}] \sigma = C \text{[} C_2 \text{]} \sigma \text{ if } E \text{[} E \text{]} \sigma = \text{false}
\]
\[
C \text{[while } E \text{ do } C \text{ end}] \sigma = \lim_{n \to \infty} C \text{[(if } E \text{ then } C \text{ else skip end) \text{]}}
\]
\[
E \text{[} V \text{]} \sigma = \sigma(V)
\]
\[
E \text{[} N \text{]} = \zeta
\]
\[
E \text{[} E_1 + E_2 \text{]} = E \text{[} E_1 \text{]} \sigma + E \text{[} E_2 \text{]} \sigma
\]
\[
E \text{[} E_1 = E_2 \text{]} \sigma = E \text{[} E_1 \text{]} \sigma = E \text{[} E_2 \text{]} \sigma
\]
Concrete Syntax of Wren
Wren

- Wren is a small imperative language that we will be using as a running example.
- The complete concrete syntax of Wren is given in the next few slides.
Concrete Syntax

program ::= \texttt{program} \texttt{identifier} \texttt{is} block

block ::= \texttt{declaration_seq} \texttt{begin} command_seq \texttt{end}

declaration_seq ::= | declaration declaration_seq

declaration ::= \texttt{var} \texttt{variable_list} : type ;

type ::= \texttt{integer} | boolean

variable_list ::= variable | variable \_ variable_list

command_seq ::= command | command ; command_seq

command ::= variable :: expr | \texttt{skip}

| \texttt{read} variable | \texttt{write} integer_expr
| \texttt{while} boolean_expr \texttt{do} command_seq \texttt{end while}
| \texttt{if} boolean_expr \texttt{then} command_seq \texttt{end if}
| \texttt{if} boolean_expr \texttt{then} command_seq \texttt{else} command_seq \texttt{end if}
Concrete Syntax...

```
expr ::= integer_expr | boolean_expr
integer_expr ::= term | integer_expr weak_op term
term ::= element | term strong_op element
element ::= numeral | variable | ( integer_expr ) | element
boolean_expr ::= boolean_term | boolean_expr or boolean_term
boolean_term ::= boolean_element
              | boolean_term and boolean_element
boolean_element ::= true | false | variable | comparison
              | not ( boolean_expr ) | ( boolean_expr )
comparison ::= integer_expr relation integer_expr
```
Concrete Syntax...

variable ::= identifier
identifier ::= letter | identifier letter | identifier digit
relation ::= <= | <= | = | > | >= | <>
weak_op ::= ± |
strong_op ::= * | /
letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m
         | n | o | p | q | r | s | t | u | v | w | x | y | z
numeral ::= digit | digit numeral
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
program binary is
  var n, p : integer;
begin
  read n; p := 2;
  while p <= n do
    p := 2 * p
  end while;
  p := p / 2;
  while p > 0 do
    if n >= p then
      write 1; n := n * p
    else
      write 0
    end if;
    p := p / 2
  end while
end
Readings and References

Acknowledgments

Some examples are taken from *Introduction to Programming Languages*, by Anthony A. Aaby,
http://burks.brighton.ac.uk/burks/pcinfo/progdocs/plbook/semantic.html

The Wren language is taken from the book *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz,