1 Formal Semantics

• 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.

2 Formal Semantics . . .

• 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.

3 Formal Semantics — Modula-2

• 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.

4 Formal Semantics — PL/I

- **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.

5 Methods

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 a 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.

6 Contextual Constraints

- A compiler performs syntactic and semantic analysis. There really isn’t a sharp distinction between the two.

- *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 restrict the programs which are considered well-formed.
7 Operational Semantics

- *Operational Semantics* specifies a language through the steps by which each program is executed.
- This is often done informally. For example, the statement \( \text{while } E \text{ do } C \) 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.

8 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.

9 Operational Sem. — Peano Arithmetic

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

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

Interpreter:

\[
I : N \rightarrow N
\]

\[
I[(n + 0)] \Rightarrow n \\
I[(m + S(n))] \Rightarrow S(I[(m + n)]) \\
I[(n \times 0)] \Rightarrow 0 \\
I[(m \times S(n))] \Rightarrow I[(m \times n) + m]
\]

where \( m, n \in \text{Nat} \)

10 Operational Sem. — Peano Arithmetic

- The rewrite rules are used to turn an expression into standard form, containing only \( S \) (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.

12 Operational Sem. — Simple...

Interpreter:

\[
\begin{align*}
I : & \quad C \times \Sigma \to \Sigma \\
\nu : & \quad E \times \Sigma \to T \cup Z
\end{align*}
\]

Semantic Equations:

\[
\begin{align*}
I(\text{skip}, \sigma) & = \sigma \\
I(V := E, \sigma) & = \sigma[V \mapsto \nu(E, \sigma)] \\
I(C_1 ; C_2, \sigma) & = E(C_2, E(C_1, \sigma)) \\
I(\text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end}, \sigma) & = I(C_1, \sigma) \text{ if } \nu(E, \sigma) = \text{true} \\
& \quad I(C_2, \sigma) \text{ if } \nu(E, \sigma) = \text{false}
\end{align*}
\]

13 Operational Sem. — Simple...

Interpreter:

\[
\begin{align*}
\text{while } E \text{ do } C \text{ end} & = \\
& \quad \text{if } E \text{ then } (C; \text{ while } E \text{ do } C \text{ end}) \text{ else skip} \\
\nu(V, \sigma) & = \sigma[V] \\
\nu(N, \sigma) & = N \\
\nu(E_1 + E_2, \sigma) & = \nu(E_1, \sigma) + \nu(E_2, \sigma) \\
\nu(E_1 = E_2, \sigma) & = \text{true} \text{ if } \nu(E, \sigma) = \nu(E, \sigma) \\
& \quad \text{false} \text{ if } \nu(E, \sigma) \neq \nu(E, \sigma)
\end{align*}
\]

14 Denotational Semantics

- 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
exec [while E do C] =
let exec-while env sto =
  let Boolean tr = evaluate [E] env sto in
  if tr then
    exec-while env (exec [C] env sto)
  else sto
in
exec-while

15 Denotational Semantics...

- 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.

16 Denotational Sem. — Peano Arithmetic

Abstract Syntax (N ∈ 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:

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

\[
D [(n + 0)] = D [n]
\]
\[
D [(m + S(n))] = D [(m + n)] + 1
\]
\[
D [(n \times 0)] = 0
\]
\[
D [(m \times S(n))] = D [(m \times n) + m]
\]

where \( m, n \in \text{Nat} \)

17 Denotational Sem. — Simple

Abstract Syntax:

- \( C \in \text{Command} \)
- \( E \in \text{Expression} \)
- \( O \in \text{Operator} \)
- \( N \in \text{Numeral} \)
• \( V \in \text{Variable} \)

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

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

\[
O ::= + \mid - \mid \ast \mid / \mid = \mid \hat{\text{i}} \mid \hat{o}
\]

18 Denotational Sem. — Simple...

Semantic Algebra:

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

Valuation Functions:

\[
\begin{align*}
C & \in C \rightarrow (S \rightarrow S) \\
E & \in E \rightarrow E \rightarrow (N \cup T)
\end{align*}
\]

19 Denotational Sem. — Simple...

\[
\begin{align*}
C [\text{skip}] \sigma & = \sigma \\
C [V := E] \sigma & = \sigma [V \leftarrow E [E]] \sigma \\
C [C_1 ; C_2] & = C [C_2] C [C_1] \\
C [\text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end}] \sigma & = C [C_1] \sigma \text{ if } E [E] \sigma = \text{true} \\
& = C [C_2] \sigma \text{ if } E [E] \sigma = \text{false} \\
C [\text{while } E \text{ do } C \text{ end}] \sigma & = \lim_{n \rightarrow \infty} C [(\text{if } E \text{ then } C \text{ else } \text{skip end})n] \sigma \\
E [V] \sigma & = \sigma (V) \\
E [N] & = \zeta \\
E [E_1 + E_2] & = E [E_1] \sigma + E [E_2] \sigma \\
E [E_1 = E_2] \sigma & = E [E_1] \sigma = E [E_2] \sigma
\end{align*}
\]

Concrete Syntax of Wren

20 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.
21 Concrete Syntax

\[
\text{program ::= program \ identifier \ in \ block}
\]

\[
\text{block ::= declaration_seq \ begin \ command_seq \ end}
\]

\[
\text{declaration_seq ::= | declaration \ declaration_seq}
\]

\[
\text{declaration ::= var \ variable_list \ : \ type \ ;}
\]

\[
\text{type ::= integer | boolean}
\]

\[
\text{variable_list ::= variable \ | variable \ and \ variable_list}
\]

\[
\text{command_seq ::= command \ | command \ and \ command_seq}
\]

\[
\text{command ::= variable ::= expr \ | \ skip}
\]

\[
\text{\ | \ \text{read \ variable \ | \ write \ integer_expr}}
\]

\[
\text{\ | \ \text{while \ boolean \ expr \ do \ command_seq \ end \ while}}
\]

\[
\text{\ | \ \text{if \ boolean \ expr \ then \ command_seq \ end \ if}}
\]

\[
\text{\ | \ \text{if \ boolean \ expr \ then \ command_seq \ else \ command_seq \ end \ if}}
\]

22 Concrete Syntax...

\[
\text{expr ::= integer_expr \ | \ boolean_expr}
\]

\[
\text{integer_expr ::= term \ | \ integer_expr \ weak_op \ term}
\]

\[
\text{term ::= element \ | \ term \ strong_op \ element}
\]

\[
\text{element ::= numeral \ | \ variable \ | \ ( \ integer_expr \ ) \ | \ element}
\]

\[
\text{boolean_expr ::= boolean_term \ | \ boolean_expr \ or \ boolean_term}
\]

\[
\text{boolean_term ::= boolean_element}
\]

\[
\text{\ | \ boolean_element \ and \ boolean_element}
\]

\[
\text{boolean_element ::= true \ | \ false \ | \ variable \ | \ comparison}
\]

\[
\text{\ | \ \text{not} \ ( \ boolean_expr \ ) \ | \ ( \ boolean_expr \ )}
\]

\[
\text{comparison ::= integer_expr \ relation \ integer_expr}
\]

23 Concrete Syntax...

\[
\text{variable ::= identifier}
\]

\[
\text{identifier ::= letter \ | \ identifier \ letter \ | \ identifier \ digit}
\]

\[
\text{relation ::= <= \ | \ <= \ | \ >= \ | \ >= \ | \ <>}
\]

\[
\text{weak_op ::= + \ |}
\]

\[
\text{strong_op ::= * \ | \ /}
\]

\[
\text{letter ::= a \ | \ b \ | \ c \ | \ d \ | \ e \ | \ f \ | \ g \ | \ h \ | \ i \ | \ j \ | \ k \ | \ l \ | \ m}
\]

\[
\text{\ | \ n \ | \ o \ | \ p \ | \ q \ | \ r \ | \ s \ | \ t \ | \ u \ | \ v \ | \ w \ | \ x \ | \ y \ | \ z}
\]

\[
\text{numeral ::= digit \ | \ digit \ numeral}
\]

\[
\text{digit ::= 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9}
\]

24 Wren — Example

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 := np
    else
        write 0
    end if;
p := p/2
end while

25 Readings and References


26 Acknowledgments

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