Chapter 3

(b) Semantics

Semantics Overview

- Syntax is about “form” and semantics about “meaning”.
- The boundary between syntax and semantics is not always clear.
- First we’ll look at issues close to the syntax end, what Sebesta calls “static semantics”, and the technique of attribute grammars.
- Then we’ll sketch three approaches to defining “deeper” semantics
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics

Why do this?

There are several reasons to want to do this.
(1) To capture some simple constraints that CFGs can not express or do so with great effort
(2) To fully specify a language
(3) To allow us to detect “semantic errors”
(4) For program verification (for critical applications)

Static Semantics

- Static semantics covers some language features that are difficult or impossible to handle in a BNF/CFG.
- It is also a mechanism for building a parser which produces a “abstract syntax tree” of it’s input.
- It can also be used to reduce some syntax trees to their values (e.g., evaluation)
- Categories attribute grammars can handle:
  - Context-free but cumbersome (e.g. type checking)
  - Non-context-free (e.g. variables must be declared before they are used)
Attribute Grammars

• Attribute Grammars (AGs) (Knuth, 1968)
• CFGs cannot describe all of the syntax of programming languages
• Additions to CFGs to carry some “semantic” info along through parse trees
• Primary value of AGs:
  • Static semantics specification
  • Compiler design (static semantics checking)

Attribute Grammar Example

In Ada we have the following rule to describe procedure definitions:

```
<proc>  -> procedure <procName> <procBody> end <procName> ;
```

But, of course, the name after “procedure” has to be the same as the name after “end”.

This is not possible to capture in a CFG (in practice) because there are too many names.

Solution: associate simple attributes with nodes in the parse tree and add a “semantic” rules or constraints to the syntactic rule in the grammar.

```
<procName>[1].string = <procName>[2].string
```

Attribute Grammars

Def: An attribute grammar is a CFG $G=(S,N,T,P)$ with the following additions:
  – For each grammar symbol $x$ there is a set $A(x)$ of attribute values.
  – Each rule has a set of functions that define certain attributes of the nonterminals in the rule.
  – Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars

Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule.

Functions of the form $S(X_0) = f(A(X_1), \ldots A(X_n))$ define synthesized attributes

Functions of the form $I(X_i) = f(A(X_0), \ldots, A(X_n))$ for $i <= j <= n$ define inherited attributes

Initially, there are intrinsic attributes on the leaves
Example: expressions of the form \( id + id \)
- \( id \)'s can be either int_type or real_type
- types of the two \( id \)'s must be the same
- type of the expression must match it's expected type

**BNF:**
\[
<expr> \rightarrow <var> + <var>
\]
\[
<var> \rightarrow id
\]

**Attributes:**
- actual_type - synthesized for \(<var>\) and \(<expr>\)
- expected_type - inherited for \(<expr>\)

---

**Attribute Grammars (continued)**

How are attribute values computed?
- If all attributes were inherited, the tree could be decorated in top-down order.
- If all attributes were synthesized, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

---

**Attribute Grammars**

**Attribute Grammar:**

   Semantic rules:
   \[
   <expr> . actual_type \leftarrow <var>[1] . actual_type
   \]
   Predicate:
   \[
   <expr> . expected_type = <expr> . actual_type
   \]

2. Syntax rule: \(<var> \rightarrow id\)
   Semantic rule:
   \[
   <var> . actual_type \leftarrow \text{lookup} (id, <var>)
   \]

---

**Attribute Grammars**

\[
<expr> . expected_type \leftarrow \text{inherited from parent}
\]
\[
<var>[1] . actual_type \leftarrow \text{lookup} (A, <var>[1])
\]
\[
<var>[2] . actual_type \leftarrow \text{lookup} (B, <var>[2])
\]
\[
\]
\[
<expr> . actual_type \leftarrow <var>[1] . actual_type
\]
\[
<expr> . actual_type =? <expr> . expected_type
\]
Attribute Grammars: another example

- Consider the following simple grammar for expressions.
- We'll show how an attribute grammar can reduce a parse tree to a value.

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow - F \\
F & \rightarrow ( E ) \\
F & \rightarrow \text{const}
\end{align*}
\]

Attribute Grammars: another example

- Each grammar symbol has a set of attributes.
  - E.g. the value of \( E_1 \) is the attribute \( E_1.\text{val} \).
- Each grammar rule has a set of rules over the symbol attributes.
  - Copy rules
  - Semantic Function rules
    - E.g. sum, quotient

Attribute Flow Example

- The figure shows the result of annotating the parse tree for \((1+3)^2\).
- Each symbol has at most one attribute shown in the corresponding box.
  - Numerical value in this example.
  - Operator symbols have no value.
- Arrows represent attribute flow.
- Note that we only use synthesized attributes (information flows up).
Static and Dynamic Semantics

- Attribute grammars add basic semantic rules to the specification of a language
  - They specify static semantics
- But they are limited to the semantic form that can be checked at compile time
- Other semantic properties cannot be checked at compile time
  - They are described using dynamic semantics
- Use to formally specify the behavior of a programming language
  - Semantic-based error detection
  - Correctness proofs

Dynamic Semantics

- No single widely acceptable notation or formalism for describing semantics.
- An approach to defining the semantics of language L is to specify a general mechanism to translate any sentence in L into a set of sentences in another language or system that we take to be well defined.
- We’ll look briefly at three approaches
  - Operational semantics -- Executing statements that represent changes in the state of a real or simulated machine
  - Axiomatic semantics -- Use predicate calculus to specify pre and post-conditions
  - Denotational semantics -- Use recursive function theory

Operational Semantics

- **Idea**: describe the meaning of a program in language L by specifying how statements effect the state of a machine, (simulated or actual) when executed.
- The change in the state of the machine (memory, registers, stack, heap, etc.) defines the meaning of the statement.
- Similar in spirit to the notion of a Turing Machine and also used informally to explain higher-level constructs in terms of simpler ones, as in:

  ```
  c statement                   operational semantics
  for(e1;e2;e3)                 e1;
  {<body>}                     loop: if e2=0 goto exit
  <body>                       <body>
  e3;                         e3;
  goto loop
  exit:
  ``

Operational Semantics

- To use operational semantics for a high-level language, a virtual machine is needed
- A **hardware** pure interpreter would be too expensive
- A **software** pure interpreter also has problems:
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent
**Operational Semantics**

*A better alternative: A complete computer simulation*

- Build a translator (translates source code to the machine code of an idealized computer)
- Build a simulator for the idealized computer

*Evaluation of operational semantics:*
- Good if used informally
- Extremely complex if used formally (e.g. VDL)

**Vienna Definition Language**

- VDL was a language developed at IBM Vienna Labs as a language for formal, algebraic definition via operational semantics.
- It was used to specify the semantics of PL/I.
- The VDL specification of PL/I was very large, very complicated, a remarkable technical accomplishment, and of little practical use.

**Axiomatic Semantics**

- Based on formal logic (first order predicate calculus)
- *Original purpose:* formal program verification
- *Approach:* Define axioms and inference rules in logic for each statement type in the language (to allow transformations of expressions to other expressions)
- The expressions are called *assertions* and are either
  - *Preconditions:* An assertion before a statement states the relationships and constraints among variables that are true at that point in execution
  - *Postconditions:* An assertion following a statement

**Logic 101**

- **Propositional logic**
  - Logical constants: true, false
  - Propositional symbols: P, Q, S, ... that are either true or false
  - Logical connectives: ∧ (and), ∨ (or), ⇒ (implies), ⇔ (is equivalent), ¬ (not) which are defined by the truth tables below.
  - Sentences are formed by combining propositional symbols, connectives and parentheses and are either true or false. e.g.: P ∨ Q ⇔ ¬(¬ P ∨ ¬ Q)

- **First order logic**
  - Variables which can range over objects in the domain of discourse
  - Quantifiers including: ∀ (for all) and ∃ (there exists)
  - Example sentences:
    - (∀ p) (∀ q) p ∨ q ⇒ ¬(¬ p ∨ ¬ q)
    - ∀ x man(x) ⇒ mortal(x)
    - ∀ x prime(x) ⇒ ∃ y prime(y) ∧ y>x

<table>
<thead>
<tr>
<th>P</th>
<th>⊤</th>
<th>¬P</th>
<th>P ∧ Q</th>
<th>P ∨ Q</th>
<th>P ⇒ Q</th>
<th>P ↔ Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊤</td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
<td>⊤</td>
</tr>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Axiomatic Semantics

- A **weakest precondition** is the least restrictive precondition that will guarantee the postcondition.

Notation:

\[
\{P\} \text{ Statement } \{Q\}
\]

(precondition)(postcondition)

Example:

\[
\{?\} a := b + 1 \quad \{a > 1\}
\]

We often need to infer what the precondition must be for a given postcondition.

One possible precondition: \(\{b > 10\}\)

Weakest precondition: \(\{b > 0\}\)

Example: Assignment Statements

Here’s how we might define a simple assignment statement of the form \(x := e\) in a programming language.

- \(\{Q_{x->E}\} x := E \{Q\}\)
- Where \(Q_{x->E}\) means the result of replacing all occurrences of \(x\) with \(E\) in \(Q\).

So from

\[
\{Q\} a := b/2 - 1 \quad \{a < 10\}
\]

We can infer that the weakest precondition \(Q\) is

\[b/2 - 1 < 10 \text{ or } b < 22\]

Program proof process:

- The postcondition for the whole program is the desired results.
- Work back through the program to the first statement.
- If the precondition on the first statement is the same as the program spec, the program is correct.

**The Rule of Consequence:**

\[
\{P\} S \{Q\}, \quad P' \Rightarrow P, \quad Q \Rightarrow Q'
\]

\[
\{P'\} S \{Q'\}
\]

An inference rule for sequences

For a sequence \(S_1; S_2\):

\[
\{P_1\} S_1 \{P_2\}
\]

\[
\{P_2\} S_2 \{P_3\}
\]

the inference rule is:

\[
\{P_1\} S_1; S_2 \{P_3\}
\]

\[
\{P_1\} S_1; S_2 \{P_3\}
\]
**Conditions**

Here’s a rule for a conditional statement

\[
\{B \land P\} S_1 \{Q\}, \quad \{\neg B \land P\} S_2 \{Q\}
\]

\{P\} if B then S_1 else S_2 \{Q\}

And an example of it’s use for the statement

\{P\} if x > 0 then y = y - 1 else y = y + 1 \{y > 0\}

So the weakest precondition P can be deduced as follows:

- The postcondition of S1 and S2 is Q.
- The weakest precondition of S1 is \(x > 0 \land y > 1\) and for S2 is \(x > 0 \land y > 0\)
- The rule of consequence and \(y > 0 \Rightarrow y > -1\) supports the conclusion
- That the weakest precondition for the entire conditional is \(y > 1\).

**Loops**

For the loop construct \{P\} while B do S end \{Q\} the inference rule is:

\[
\{I \land B\} \quad S \quad \{I\}
\]

\{I\} while B do S \{I \land \neg B\}

where I is the loop invariant, a proposition necessarily true throughout the loop’s execution.

**Loop Invariants**

A loop invariant I must meet the following conditions:

1. \(P \Rightarrow I\) (the loop invariant must be true initially)
2. \(\{I\} B \{I\}\) (evaluation of the Boolean must not change the validity of I)
3. \(\{I \land B\} \quad S \quad \{I\}\) (I is not changed by executing the body of the loop)
4. \(\{I \land \neg B\} \Rightarrow Q\) (if I is true and B is false, Q is implied)
5. The loop terminates (this can be difficult to prove)

- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.
- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

**Evaluation of Axiomatic Semantics**

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs
- It is much less useful for language users and compiler writers
Denotational Semantics

• A technique for describing the meaning of programs in terms of mathematical functions on programs and program components.

• Programs are translated into functions about which properties can be proved using the standard mathematical theory of functions, and especially domain theory.

• Originally developed by Scott and Strachey (1970) and based on recursive function theory.

• The most abstract semantics description method.

Denotational Semantics

• The process of building a denotational specification for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

• The meaning of language constructs are defined by only the values of the program's variables.

Denotational Semantics (continued)

The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions.

• The state of a program is the values of all its current variables.

\[ s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\} \]

• Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable.

\[ \text{VARMAP}(i, s) = v \]

Example: Decimal Numbers

\[ \langle \text{dec\_num} \rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

\[ \langle \text{dec\_num} \rangle \mid \langle \text{dec\_num} \rangle \langle 0|1|2|3|4|5|6|7|8|9 \rangle \]

\[ M_{\text{dec}}('0') = 0, \ M_{\text{dec}}('1') = 1, \ldots, M_{\text{dec}}('9') = 9 \]

\[ M_{\text{dec}}(\langle\text{dec\_num}\rangle '0') = 10 \times M_{\text{dec}}(\langle\text{dec\_num}\rangle) \]

\[ M_{\text{dec}}(\langle\text{dec\_num}\rangle '1') = 10 \times M_{\text{dec}}(\langle\text{dec\_num}\rangle) + 1 \]

\[ \ldots \]

\[ M_{\text{dec}}(\langle\text{dec\_num}\rangle '9') = 10 \times M_{\text{dec}}(\langle\text{dec\_num}\rangle) + 9 \]
Expressions

\[ M_e(<\text{expr}>, s) \triangleq \begin{cases} 
\text{case } <\text{expr}> \text{ of} \\
<\text{dec_num}> & \Rightarrow M_{\text{dec}}(<\text{dec_num}>, s) \\
<\text{var}> & \Rightarrow \\
\text{if } \text{VARMAP}(<\text{var}>, s) = \text{undef} \\
\text{then } \text{error} \\
\text{else } \text{VARMAP}(<\text{var}>, s) \\
<\text{binary_expr}> & \Rightarrow \\
\text{if } (M_e(<\text{binary_expr}.<\text{left_expr}>, s) = \text{undef} \\
\text{OR } M_e(<\text{binary_expr}.<\text{right_expr}>, s) = \text{undef}) \\
\text{then } \text{error} \\
\text{else } \\
\text{if } (<\text{binary_expr}.<\text{operator}> = '+' \text{ then} \\
M_e(<\text{binary_expr}.<\text{left_expr}>, s) + \\
M_e(<\text{binary_expr}.<\text{right_expr}>, s) \\
\text{else } M_e(<\text{binary_expr}.<\text{left_expr}>, s) * \\
M_e(<\text{binary_expr}.<\text{right_expr}>, s) \\
\end{cases} \]

Assignment Statements

\[ M_a(x := E, s) \triangleq \begin{cases} 
\text{if } M_e(E, s) = \text{error} \\
\text{then } \text{error} \\
\text{else } s' = \{<i_1',v_1'>,<i_2',v_2'>,...,<i_n',v_n'>\}, \\
\text{where for } j = 1, 2, ..., n, \\
v_{ij}' = \text{VARMAP}(i_j, s) \text{ if } i_j <> x \\
= M_e(E, s) \text{ if } i_j = x \\
\end{cases} \]

Logical Pretest Loops

\[ M_l(\text{while } B \text{ do } L, s) \triangleq \begin{cases} 
\text{if } M_b(B, s) = \text{undef} \\
\text{then } \text{error} \\
\text{else if } M_b(B, s) = \text{false} \\
\text{then } s \\
\text{else if } M_o(L, s) = \text{error} \\
\text{then } \text{error} \\
\text{else } M_l(\text{while } B \text{ do } L, M_o(L, s)) \\
\end{cases} \]

Logical Pretest Loops

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
- Recursion, when compared to iteration, is easier to describe with mathematical rigor
Denotational Semantics

*Evaluation of denotational semantics:*
  - Can be used to prove the correctness of programs
  - Provides a rigorous way to think about programs
  - Can be an aid to language design
  - Has been used in compiler generation systems

Summary

This chapter covered the following
  - Backus-Naur Form and Context Free Grammars
  - Syntax Graphs and Attribute Grammars
  - Semantic Descriptions: Operational, Axiomatic and Denotational