Semantic Analysis

In Text: Chapter 3

Outline

• Static semantics
  – Attribute grammars
• Dynamic semantics
  – Operational semantics
  – Denotational semantics

Syntax vs. Semantics

• Syntax concerns the form of a valid program
• Semantics concerns its meaning
• Meaning of a program is important
  – It allows us to enforce rules, such as type consistency, which go beyond the form
  – It provides the information needed to generate an equivalent output program

Two types of semantic rules

• Static semantics
• Dynamic semantics

Static Semantics

• There are some characteristics of the structure of programming languages that are difficult or impossible to describe with BNF
  – E.g., type compatibility: a floating-point value cannot be assigned to an integer type variable, but the opposite is legal

Static Semantics

• The static semantics of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs
  – Syntax rather than semantics
• Many static semantic rules of a language state its type constraints
Dynamic semantics

- It describes the meaning of expressions, statements, and program units
- Programmers need dynamic semantics to know precisely what statements of a language do
- Compiler writers need define the semantics of the languages for which they are writing compilers

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - Semantic analysis
  - (intermediate) code generation

Role of Semantic Analysis

- The principal job of the semantic analyzer is to enforce static semantics
  - Constructs a syntax tree (usually first)
  - Performs analysis of information that is gathered
  - Uses that information later during code generation

Conventional Semantic Analysis

- Compile-time analysis and run-time "actions" that enforce language-defined semantics
  - Static semantic rules are enforced at compile time by the compiler
    - Type checking
  - Dynamic semantic rules are enforced at runtime by the compiler-generated code
    - Bounds checking

Attribute Grammar

- A device used to describe more of the structure of a programming language than can be described with a context-free grammar
- It provides a formal framework for decorating parse trees
- An attribute grammar is an extension to a context-free grammar

STATIC SEMANTICS
Attribute Grammar

- The extension includes
  - Attributes
  - Attribute computation functions
  - Predicate functions

Attributes

- Associated with each grammar symbol $X$ is a set of attributes $A(X)$. The set $A(X)$ consists of two disjoint sets: $S(X)$ and $I(X)$

Attributes

- $S(X)$: synthesized attributes, which are used to pass semantic information bottom-up in a parse tree

Attributes

- $I(X)$: inherited attributes, which pass semantic information down or across a tree. Similar to variables because they can also have values assigned to them

Intrinsic Attributes

- Synthesized attributes of leaf nodes whose values are determined outside the parse tree
  - E.g., the type of a variable can come from the symbol table
  - Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values

A Running Example

- Context-Free Grammar (CFG)
  
  $$<\text{assign}> \rightarrow <\text{var}> = <\text{expr}>$$
  $$<\text{expr}> \rightarrow <\text{var}> + <\text{expr}>$$
  $$<\text{expr}> \rightarrow <\text{var}>$$
  $$<\text{var}> \rightarrow A \mid B \mid C$$

- Note:
  - It only focuses on potential structured sequences of tokens
  - It says nothing about the meaning of any particular program
Semantic Functions
• Specify how attribute values are computed for \( S(X) \) and \( I(X) \)

Semantic Functions
• For a rule \( X_0 \rightarrow X_1 \ldots X_n \), the synthesized attributes of \( X_0 \) are computed with semantic functions of the form \( S(X_0) = f(A(X_1), \ldots, A(X_n)) \)

Semantic Functions
• Inherited attributes of symbols \( X_j \), \( 1 \leq j \leq n \), are computed with a semantic function of the form \( I(X_j) = f(A(X_0), \ldots, A(X_n)) \)
• To avoid circularity, inherited attributes are often restricted to functions of the form \( I(X_j) = f(A(X_0), \ldots, A(X_{j-1})) \)

Predicate Functions
• A predicate function has the form of a Boolean expression on the union of the attribute set \( (A(X_0), \ldots, A(X_n)) \), and a set of literal attribute values
• A false predicate function value indicates a violation of the syntax or static semantic rules

An Attribute Grammar Example
• \textit{actual	extunderscore type} (a synthesized attribute)
  – It is used to store the actual type, int or real, of a variable or expression
  – For each concrete variable, the \textit{actual	extunderscore type} is intrinsic
  – For expressions and assignments, the attribute is determined by the actual types of children nodes

An Attribute Grammar Example (Cont’d)
• \textit{expected	extunderscore type} (an inherited attribute)
  – Associated with the nonterminal \textless expr\textgreater
  – It is used to store the expected type, either int or real
  – It is determined by the type of the variable on the left side of the assignment statement
An Attribute Grammar Example (Cont'd)

1. Syntax rule: `<assign> -> <var> = <expr>
   Semantic rule: `<expr>.expected_type <- `<var>.actual_type
   Semantic rule: `<expr>.actual_type <- `<var>[2].actual_type = int) and (`var[3].actual_type = int)
   then int
   else real
   end if
   Predicate: `<expr>.actual_type == `<expr>.expected_type

Another Example: Constant Expressions

- CFG
  - `E -> `E + `T
  - `E -> `E - `T
  - `E -> `T
  - `T -> `T * `F
  - `T -> `T / `F
  - `T -> `- `F
  - `F -> ( `E )
  - `F -> const

Note:
- Says nothing about the meaning of any particular program
- Conveys only potential structured sequence of tokens

Example Attribute Grammar

- Attribute: val
- Attribute Grammar
  - `E1 -> `E2 + `T
  - `E1 -> `E2 - `T
  - `E1 -> `T
  - `T1 -> `T2 * `F
  - `T1 -> `T2 / `F
  - `T1 -> `- `F
  - `F -> ( `E )
  - `F -> const

  `E1.val = `E2.val + `T.val
  `E1.val = `E2.val - `T.val
  `E1.val = `T.val
  `T1.val = `T2.val * `F.val
  `T1.val = `T2.val / `F.val
  `T1.val = `- `F2.val
  `F.val = `E.val
  `F.val = `C.val

Evaluating Attributes

- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree
- If all attributes are inherited, the evaluation process can be done in a top-down order
- Alternatively, if all attributes are synthesized, the evaluation can proceed in a bottom-up order

An Example Parse Tree

- We have both inherited and synthesized attributes. In what direction should we proceed the computation?
An Example Parse Tree

<table>
<thead>
<tr>
<th>Rule</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>&lt;var&gt;.actual_type</code> ← look-up(A) (R4)</td>
<td></td>
</tr>
<tr>
<td>2. <code>&lt;var&gt;.actual_type</code> ← look-up(B) (R4)</td>
<td></td>
</tr>
<tr>
<td>3. <code>&lt;var&gt;.actual_type</code> ← look-up(A) (R1)</td>
<td></td>
</tr>
<tr>
<td>4. <code>&lt;var&gt;.actual_type</code> ← look-up(B) (R4)</td>
<td></td>
</tr>
<tr>
<td>5. <code>&lt;expr&gt;.actual_type</code> ← real (R2)</td>
<td></td>
</tr>
<tr>
<td>6. <code>&lt;expr&gt;.expected_type</code> ← <code>&lt;expr&gt;.actual_type</code> is TRUE (R2)</td>
<td></td>
</tr>
</tbody>
</table>

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

Decoration of a parse tree for the val attribute evaluation of \((1 + 3) \times 2\)

E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → – F
F → ( E )
F → const

Attribute Evaluation Order

- Determining attribute evaluation order for any attribute grammar is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → – F
F → ( E )
F → const