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

N. Meng, S. Arthur 7

Role of Semantic Analysis

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

N. Meng, S. Arthur 8

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

N. Meng, S. Arthur 9

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

N. Meng, S. Arthur 10

STATIC SEMANTICS

N. Meng, S. Arthur 11

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

N. Meng, S. Arthur 12

Attribute Grammar

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

A Running Example

- Context-Free Grammar (CFG)

```
<assign> -> <var> = <expr>
<expr>   -> <var> + <var>
<expr>   -> <var>
<var>    -> A | B | C
```

- Note:
  - It only focuses on potential structured sequences of tokens
  - It says nothing about the meaning of any particular program

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

Semantic Functions

- Specify how attribute values are computed for S(X) and I(X)

Semantic Functions

- For a rule X_0->X_1…X_n, the synthesized attributes of X_0 are computed with semantic functions of the form $S(X_0) = f(A(X_1), …, A(X_n))$

Semantic Functions

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

Predicate Functions

- A predicate function has the form of a Boolean expression on the union of the attribute set $\{A(X_0), …, A(X_n)\}$, 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

- *actual_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 actual_type is intrinsic
  - For expressions and assignments, the attribute is determined by the actual types of children nodes

An Attribute Grammar Example (Cont'd)

- *expected_type (an inherited attribute)*
  - Associated with the nonterminal <expr>
  - 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
2. Syntax rule: <expr> -> <var>[2] + <var[3]
   Semantic rule: <expr>.actual_type <-
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type

An Attribute Grammar Example (Cont'd)

3. Syntax rule: <expr> -> <var>
   Semantic rule: <expr>.actual_type <- <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> -> A | B | C
   Semantic rule: <var>.actual_type <- look-up(<var>.string)

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

Another Example: Constant Expressions

- CFG

```
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → - 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

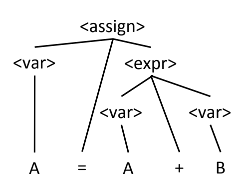
- Attribute: val
- Attribute Grammar

$E_1 \rightarrow E_2 + T$	E1.val = E2.val + T.val
$E_1 \rightarrow E_2 - T$	E1.val = E2.val - T.val
$E \rightarrow T$	E.val = T.val
$T_1 \rightarrow T_2 * F$	T1.val = T2.val * F.val
$T_1 \rightarrow T_2 / F$	T1.val = T2.val / F.val
$T \rightarrow F$	T.val = F.val
$F_1 \rightarrow - F_2$	F1.val = - F2.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ const	F.val = C.val

Evaluating Attributes

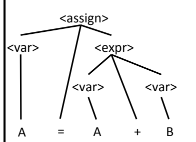
- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree
- If all attributes are inherited, the evaluation process can be done in a top-down order
- Alternatively, if all attributes are synthesized, the evaluation can proceed in a bottom-up order

An Example Parse Tree



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

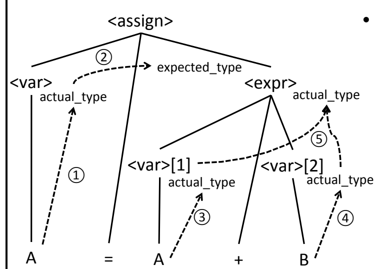
An Example Parse Tree



<expr>.expected_type <- <var>.actual_type
<expr>.actual_type <- if (<var>[2].actual_type = int) and (<var>[3].actual_type = int) then int else real end if <expr>.actual_type == <expr>.expected_type
<expr>.actual_type <- <var>.actual_type <expr>.actual_type == <expr>.expected_type
<var>.actual_type <- look-up(<var>.string) The look-up function looks up a given variable name in the symbol table and returns the variable's type

1. <var>.actual_type <- look-up(A) (R4)
2. <expr>.expected_type <- <var>.actual_type (R1)
3. <var>.actual_type <- look-up(A) (R4)
4. <var>.actual_type <- look-up(B) (R4)
5. <expr>.actual_type <- real (R2)
6. <expr>.expected_type == <expr>.actual_type is TRUE (R2)

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

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

```
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → - F
F → ( E )
F → const
```