

Semantic Analysis

In Text: Chapter 3

Outline

- Static semantics
 - Attribute grammars
- Dynamic semantics
 - Operational semantics
 - Denotational semantics

N. Meng, S. Arthur

2

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

N. Meng, S. Arthur

3

Two types of semantic rules

- Static semantics
- Dynamic semantics

N. Meng, S. Arthur

4

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

N. Meng, S. Arthur

5

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

N. Meng, S. Arthur

6

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

N. Meng, S. Arthur

13

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

N. Meng, S. Arthur

14

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 $I(X)$

N. Meng, S. Arthur

15

Attributes

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

N. Meng, S. Arthur

16

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

N. Meng, S. Arthur

17

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

N. Meng, S. Arthur

18

Semantic Functions

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

N. Meng, S. Arthur

19

Semantic Functions

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

N. Meng, S. Arthur

20

Semantic Functions

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

N. Meng, S. Arthur

21

Predicate Functions

- A predicate function has the form of a Boolean expression on the union of the attribute set $\{A(X_0), \dots, A(X_n)\}$, and a set of literal attribute values
- A false predicate function value indicates a violation of the syntax or static semantic rules

N. Meng, S. Arthur

22

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

N. Meng, S. Arthur

23

An Attribute Grammar Example (Cont'd)

- *expected_type* (an inherited attribute)
 - Associated with the nonterminal $\langle \text{expr} \rangle$
 - 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

N. Meng, S. Arthur

24

An Attribute Grammar Example (Cont'd)

- Syntax rule: $\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$
 Semantic rule: $\langle \text{expr} \rangle.\text{expected_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
- Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[2] + \langle \text{var} \rangle[3]$
 Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow$
 if $(\langle \text{var} \rangle[2].\text{actual_type} = \text{int})$ and
 $(\langle \text{var} \rangle[3].\text{actual_type} = \text{int})$
 then int
 else real
 end if
 Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$

An Attribute Grammar Example (Cont'd)

- Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle$
 Semantic rule: $\langle \text{expr} \rangle.\text{actual_type} \leftarrow \langle \text{var} \rangle.\text{actual_type}$
 Predicate: $\langle \text{expr} \rangle.\text{actual_type} == \langle \text{expr} \rangle.\text{expected_type}$
- Syntax rule: $\langle \text{var} \rangle \rightarrow A | B | C$
 Semantic rule: $\langle \text{var} \rangle.\text{actual_type} \leftarrow \text{look-up}(\langle \text{var} \rangle.\text{string})$
 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
 - $E \rightarrow E + T$
 - $E \rightarrow E - T$
 - $E \rightarrow T$
 - $T \rightarrow T * F$
 - $T \rightarrow T / F$
 - $T \rightarrow F$
 - $F \rightarrow - F$
 - $F \rightarrow (E)$
 - $F \rightarrow \text{const}$
- Note:
 - Says nothing about the meaning of any particular program
 - Conveys only potential structured sequence of tokens

Example Attribute Grammar

- Attribute: val
- Attribute Grammar

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

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
  
```

- <var>.actual_type <- look-up(A) (R4)
- <expr>.expected_type <- <var>.actual_type (R1)
- <var>.actual_type <- look-up(A) (R4)
- <var>.actual_type <- look-up(B) (R4)
- <expr>.actual_type <- real (R2)
- <expr>.expected_type == <expr>.actual_type is TRUE (R2)

N. Meng, S. Arthur 31

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

N. Meng, S. Arthur 32

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

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

N. Meng, S. Arthur 33

Attribute Grammar for Constant Expressions based on LL(1) CFG

- $E \rightarrow T TT$
 $\triangleright TT.st := T.val$ $\triangleright E.val := TT.val$
- $TT_1 \rightarrow + T TT_2$
 $\triangleright TT_2.st := TT_1.st + T.val$ $\triangleright TT_1.val := TT_2.val$
- $TT_1 \rightarrow - T TT_2$
 $\triangleright TT_2.st := TT_1.st - T.val$ $\triangleright TT_1.val := TT_2.val$
- $TT \rightarrow \epsilon$
 $\triangleright TT.val := TT.st$
- $T \rightarrow F FT$
 $\triangleright FT.st := F.val$ $\triangleright T.val := FT.val$
- $FT_1 \rightarrow * F FT_2$
 $\triangleright FT_2.st := FT_1.st \times F.val$ $\triangleright FT_1.val := FT_2.val$
- $FT_1 \rightarrow / F FT_2$
 $\triangleright FT_2.st := FT_1.st \div F.val$ $\triangleright FT_1.val := FT_2.val$
- $FT \rightarrow \epsilon$
 $\triangleright FT.val := FT.st$
- $F_1 \rightarrow - F_2$
 $\triangleright F_1.val := - F_2.val$
- $F \rightarrow (E)$
 $\triangleright F.val := E.val$
- $F \rightarrow const$
 $\triangleright F.val := const.val$

N. Meng, S. Arthur 34

Attribute Grammar for CE LL(1) CFG

- Attributes
 - *st*: subtotal attribute to record intermediate evaluation result so far
 - *val*: value attribute to copy the right-most leaf back up to the root

N. Meng, S. Arthur 35

Decoration of parse tree for (1 + 3) * 2

N. Meng, S. Arthur 36