Programming Languages

Lecture 7: Semantic Analysis

Benjamin J. Keller

Department of Computer Science, Virginia Tech
Outline

- Semantics versus Syntax
- Role of Semantic Analyzer
- Attribute Grammars
- Attribute Flow
- Action Routines
- Tree Grammars
Syntax versus Semantics

- *Syntax* — determines valid form of program
- *Semantics* — behavior of valid program
- Convention is that syntax is what can be specified by CFG
- Doesn’t match intuition — some things that seem to be syntax aren’t definable in CFG
  
  Ex. number of arguments in function call

- In practice, anything that requires compiler to compare constructs separated with other code, or to count items or nested structures are semantics.
Semantics

- **Static** semantics — can be analyzed at compile-time
- **Dynamic** semantics — analyzed at runtime
  - Division by zero
  - Array bounds checks
- Not a clear distinction or boundary
- Theory says that while some problems can be found at compile-time, not all can.
- So, must have runtime semantic checks
Semantic Analyzer

- Semantic analyzer
  - Determines meaning of program
  - Enforces semantic rules

- Role in compiler varies
  - Strict boundary between parsing, analysis and synthesis
  - Generally some interleaving of three activities
  - Some compilers perform semantic analysis on intermediate forms
Attribute Grammars

- “Decorated” context free grammar
- Associate attributes with nonterminals of grammar
- Associate rule with each production
  \[ E_1 \rightarrow E_2 + T \]
  \[ \triangleright E_1.val := \text{sum}(E_2.val, T.val) \]
- Must uniquely identify nonterminal occurrences
Attribute Grammar Rules

• Rule can invoke \textit{semantic functions}

\[
E_1 \rightarrow E_2 + T \\
\triangleright E_1.val := \text{sum}(E_2.val, T.val)
\]

• Rule can copy values

\[
E \rightarrow T \\
\triangleright E.val := T.val
\]
Kinds of Attributes

- **Synthesized attribute** — value only computed when symbol is on left-hand side of production
  - Attributes can be computed independently of context
  - S-attributed grammar has only synthesized attributes

- **Inherited attribute** — value computed in productions where symbol is on right-hand side
  - Attributes computed using context
  - Cannot avoid these in semantic analysis
Example: Binary Numbers with Fractions

<table>
<thead>
<tr>
<th>Nonterminals</th>
<th>Synthesized Attribute(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>val</td>
</tr>
<tr>
<td>string</td>
<td>val, len</td>
</tr>
<tr>
<td>bit</td>
<td>val</td>
</tr>
</tbody>
</table>
Grammar for Binary Numbers

(1) \[ \text{num} \rightarrow \text{string}_1 \cdot \text{string}_2 \]
\[ \triangledown \text{num}.\text{val} := \text{string}_1.\text{val} + \text{string}_2.\text{val}/2^{\text{string}_2.\text{len}} \]

(2) \[ \text{num} \rightarrow \text{string} \]
\[ \triangledown \text{num}.\text{val} := \text{string}.\text{val} \]

(3) \[ \text{string}_0 \rightarrow \text{string}_1 \cdot \text{bit} \]
\[ \triangledown \text{string}_0.\text{val} := 2 \cdot \text{string}_1.\text{val} + \text{bit}.\text{val} \]
\[ \triangledown \text{string}_0.\text{len} := \text{string}_1.\text{len} + 1 \]

(4) \[ \text{string} \rightarrow \text{bit} \]
\[ \triangledown \text{string}.\text{val} := \text{bit}.\text{val} \]
\[ \triangledown \text{string}.\text{len} := 1 \]

(5) \[ \text{bit} \rightarrow 0 \]
\[ \triangledown \text{bit}.\text{val} := 0 \]

(6) \[ \text{bit} \rightarrow 1 \]
\[ \triangledown \text{bit}.\text{val} := 1 \]
Parse Tree for 1101.01

```
num

string .

string bit

string bit 1

string bit 0

bit 1

1

bit 0

bit 1

1

num.val = 13.25
```
Example: Using Inherited Attributes

<table>
<thead>
<tr>
<th>Nonterminals</th>
<th>Synthesized Attribute(s)</th>
<th>Inherited Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>val</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>val, len</td>
<td>pos</td>
</tr>
<tr>
<td>bit</td>
<td>val</td>
<td>pos</td>
</tr>
</tbody>
</table>
Grammar

(1) \( num \rightarrow string_1 \cdot string_2 \)
   \( \triangleright num.val := string_1.val + string_2.val \)
   \( \triangleright string_1.pos := 0 \)
   \( \triangleright string_2.pos := -1 \)

(2) \( num \rightarrow string \)
   \( \triangleright num.val := string.val \)
   \( \triangleright string.pos := 0 \)
Grammar (cont)

(3) \( \text{string}_0 \rightarrow \text{string}_1 \text{ bit} \)

- \( \text{string}_0.\text{val} := \text{string}_1.\text{val} + \text{bit}.\text{val} \)
- \( \text{string}_0.\text{len} := \text{string}_1.\text{len} + 1 \)
- if \( \text{string}_0.\text{pos} \geq 0 \) then
  - \( \text{string}_1.\text{pos} := \text{string}_0.\text{pos} + 1 \)
  - \( \text{bit}.\text{pos} := \text{string}_0.\text{pos} \)
else
  - \( \text{string}_1.\text{pos} := \text{string}_0.\text{pos} - 1 \)
  - \( \text{bit}.\text{pos} := -\text{string}_0.\text{len} \)
(4) \[string \rightarrow bit\]
\[\begin{align*}
\& \quad string.val := bit.val \\
\& \quad string.len := 1 \\
\& \quad \text{if } string.pos \geq 0 \text{ then} \\
\& \quad \quad bit.pos := string.pos \\
\& \quad \text{else} \\
\& \quad \quad bit.pos := -1
\end{align*}\]

(5) \[bit \rightarrow 0\]
\[\begin{align*}
\& \quad bit.val := 0
\end{align*}\]

(6) \[bit \rightarrow 1\]
\[\begin{align*}
\& \quad bit.val := 2^{bit.pos}
\end{align*}\]
Parse Tree for 110.101

\[
\text{num.val} = 6.625
\]
**Attribute Flow**

- Pattern of information flow between attributes
- Necessary flow determined by language and parsing technique
- Example: arithmetic expressions
  - Can define S-attributed grammar from SLR grammar
  - LL(1) equivalent must have inherited attributes
L-Attributed Grammars

- Attribute $A.s$ depends on attribute $B.t$ if $B.t$ is passed to a semantic function that returns a value for $A.s$

- A grammar is L-attributed if
  1. each synthesized attribute of a left-hand side symbol depends only on inherited attributes of that symbol, or on attributes of the symbols on the right-hand side of the production; and
  2. each inherited attribute of a right-hand side symbol depends only on inherited attributes of the left-hand side symbol or on attributes of symbols to its left in the right-hand side.
Parsing and Attribute Flow

- S-attributed grammars are those that can be evaluated *on-the-fly* with an LR parse
- L-attributed grammars are those that can be evaluated *on-the-fly* with an LL parse
- Evaluating *on-the-fly* refers to interleaving parsing and attribute evaluation
- *One-pass* compiler fully interleaves parsing and code generation
Action Routines

- Semantic function that compiler executes during parsing
- Used in parser generators
- In LL parse may occur anywhere in production
  - Only use production if know it is correct
  - Example: ANTLR
- In LR parse must occur at end of production
  - Rationale: don’t know if production applies until see full rhs
  - Example: YACC and variants
ANTLR Grammar

arg_lst[SymbolTable& st] > [list<Decl> 1] :
    nme:IDENTIFIER "::" typ:IDENTIFIER
    << if ($st.isDefinedType($typ->getText()))
        $l.push_back(Decl($nme->getText(),$typ->getText()));
    >>
    ( ","
    nme2:IDENTIFIER "::" typ2:IDENTIFIER
    << if ($st.isDefinedType($typ2->getText()))
        $l.push_back(Decl($nme2->getText(),$typ2->getText()));
    >>
    )*
;
YACC Grammar

```
%token NAME NUMBER
%
statement: NAME '=' expression
    | expression    { printf("= %d\n", $1); }
    ;

expression: expression '+' NUMBER    { $$ = $1 + $3; }
    | expression '-' NUMBER    { $$ = $1 - $3; }
    | NUMBER             { $$ = $1; }
    ;
```

Analysis of Abstract Syntax Trees

- Common for parser to generate AST for analysis
- Describe structure of AST as tree grammar
- Form attribute grammar from tree grammar instead of CFG
- Allows analysis of AST