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

$$
\begin{array}{rcl}
E_1 & \rightarrow & E_2 + T \\
 & \triangleright & E_1.val := sum(E_2.val, T.val)
\end{array}
$$

- Must uniquely identify nonterminal occurrences

Attribute Grammar Rules

- Rule can invoke *semantic functions*

$$E_1 \rightarrow E_2 + T$$
$$\triangleright \quad E_1.val := sum(E_2.val, T.val)$$

- Rule can copy values

$$E \rightarrow T$$
$$\triangleright \quad E.val := T.val$$

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

Nonterminals	Synthesized Attribute(s)
num	*val*
string	*val, len*
bit	*val*

Grammar for Binary Numbers

(1) $num \rightarrow string_1 \;.\; string_2$

$\triangleright\ num.val := string_1.val + string_2.val/2^{string_2.len}$

(2) $num \rightarrow string$

$\triangleright\ num.val := string.val$

(3) $string_0 \rightarrow string_1\ bit$

$\triangleright\ string_0.val := 2 \cdot string_1.val + bit.val$

$\triangleright\ string_0.len := string_1.len + 1$

(4) $string \rightarrow bit$

$\triangleright\ string.val := bit.val$

$\triangleright\ string.len := 1$

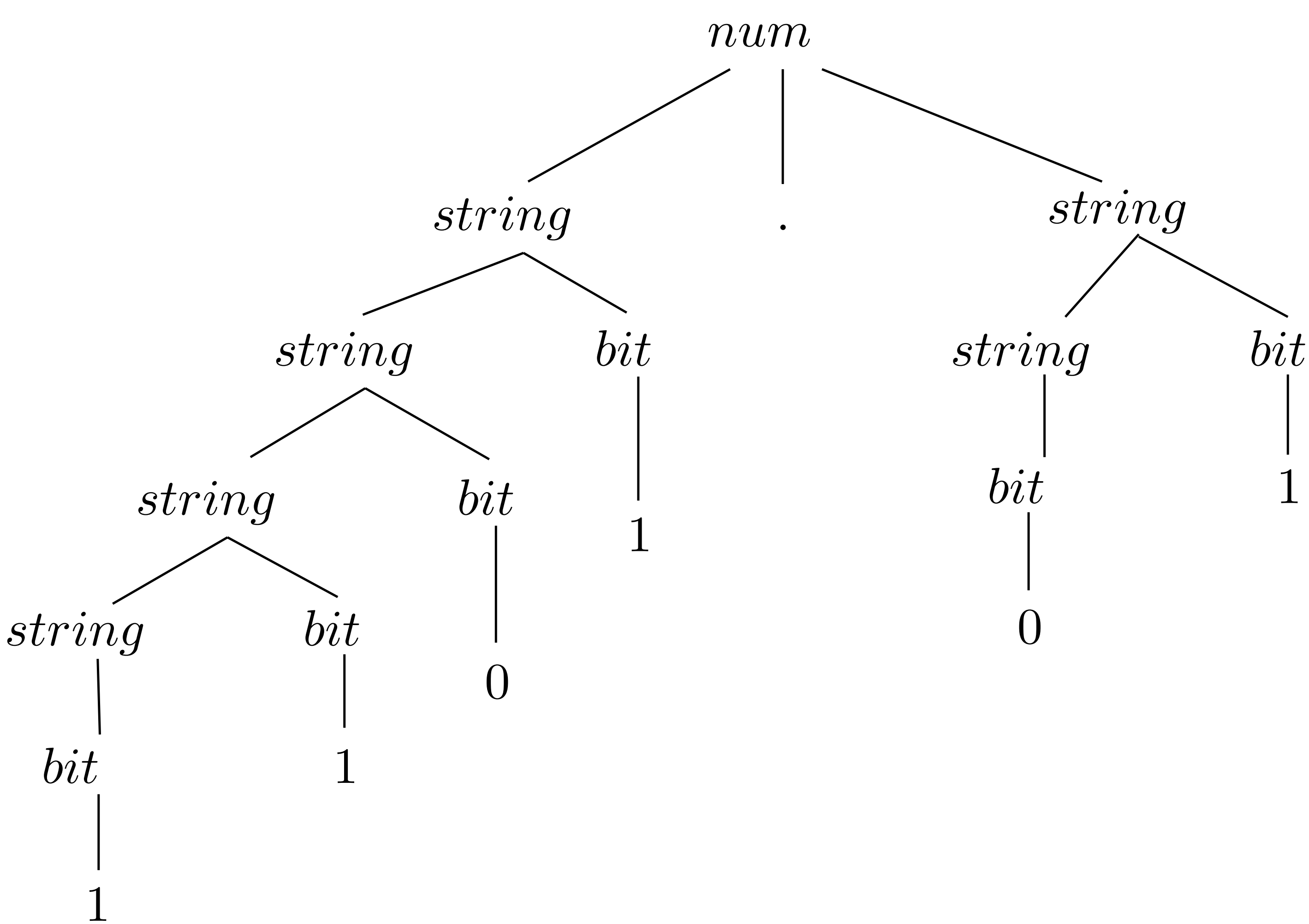
(5) $bit \rightarrow 0$

$\triangleright\ bit.val := 0$

(6) $bit \rightarrow 1$

$\triangleright\ bit.val := 1$

Parse Tree for 1101.01



$num.val = 13.25$

Example: Using Inherited Attributes

Nonterminals	Synthesized Attribute(s)	Inherited Attributes
num	*val*	
string	*val, len*	*pos*
bit	*val*	*pos*

Grammar

(1) $num \rightarrow string_1\ .\ string_2$

$\triangleright\ num.val := string_1.val + string_2.val$

$\triangleright\ string_1.pos := 0$

$\triangleright\ string_2.pos := -1$

(2) $num \rightarrow string$

$\triangleright\ num.val := string.val$

$\triangleright\ string.pos := 0$

Grammar (cont)

(3) $string_0 \rightarrow string_1\ bit$

$\triangleright$ $string_0.val := string_1.val + bit.val$

$\triangleright$ $string_0.len := string_1.len + 1$

$\triangleright$ if $string_0.pos \geq 0$ then

$string_1.pos := string_0.pos + 1$

$bit.pos := string_0.pos$

else

$string_1.pos := string_0.pos - 1$

$bit.pos := -string_0.len$

Grammar (cont)

(4) $string \rightarrow bit$

- $\triangleright$ $string.val := bit.val$
- $\triangleright$ $string.len := 1$
- $\triangleright$ if $string.pos \geq 0$ then

  $bit.pos := string.pos$

  else

  $bit.pos := -1$

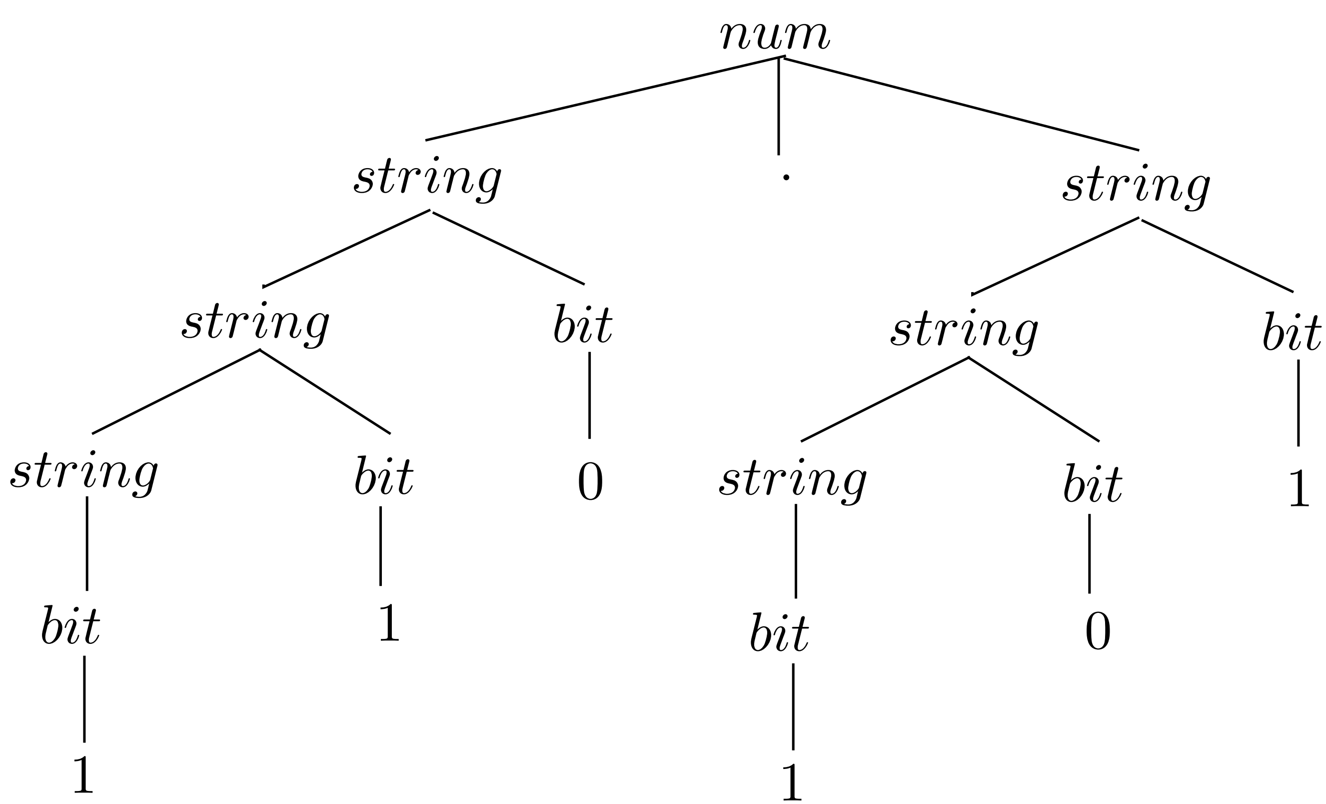
(5) $bit \rightarrow 0$

- $\triangleright$ $bit.val := 0$

(6) $bit \rightarrow 1$

- $\triangleright$ $bit.val := 2^{bit.pos}$

Parse Tree for 110.101



$num.val = 6.625$

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> l] :
    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