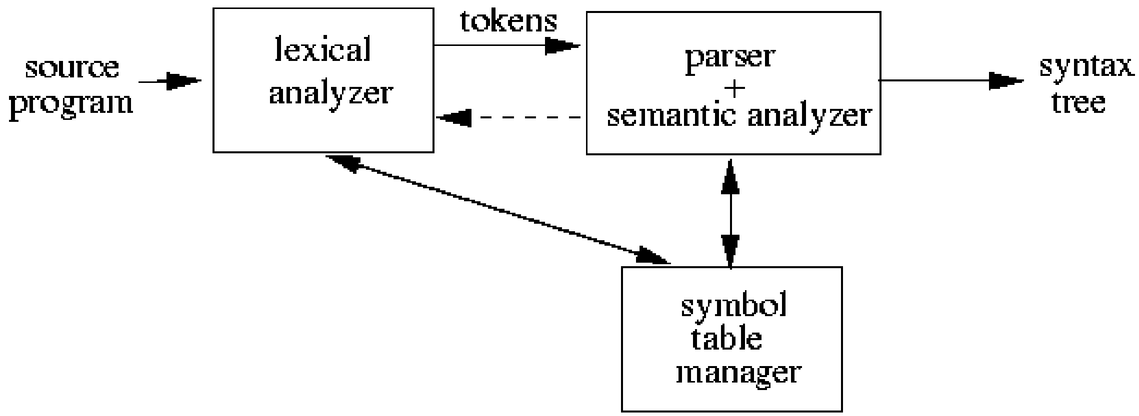
Syntax Analysis (Parsing)

19CSE401 Compiler Design

Overview



Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.

Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

Grammars

- Every programming language has precise **grammar rules** that describe the **syntactic structure** of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc
- Grammars are easy to understand, and **parsers** for programming languages can be constructed automatically from certain classes of grammars
- **Parsers** or **syntax analyzers** are generated for a particular grammar
- **Context-free grammars** are usually used for syntax specification of programming languages

What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated is based on context-free grammars.
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing **context-free languages** (like FSA for RL)

Context-free Grammars

- A *context-free grammar* for a language specifies the syntactic structure of programs in that language.
- Components of a grammar:
  - a finite set of tokens (obtained from the scanner);
  - a set of variables representing "related" sets of strings, e.g., *declarations*, *statements*, expressions.
  - a set of rules that show the structure of these strings.
  - an indication of the "top-level" set of strings we care about.

Context-free Grammars: Definition

Formally, a context-free grammar G is a 4-tuple

G = (N, T, P, S), where:

- N: Finite set of non-terminals
- T: Finite set of terminals
- S ∈ N: The start symbol
- P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)*$

- Usually, only P is specified and the first production corresponds to that of the start symbol
- Ex:

(1)	(2)	(3)	(4)
$E \rightarrow E + E$	$S \rightarrow 0S0$	$S \rightarrow aSb$	$S \rightarrow aB \mid bA$
$E \rightarrow E * E$	$S \rightarrow 1S1$	$S \rightarrow \epsilon$	$A \rightarrow a \mid aS \mid bAA$
$E \rightarrow (E)$	$S \rightarrow 0$		$B \rightarrow b \mid bS \mid aBB$
$E \rightarrow id$	$S \rightarrow 1$		
	$S \rightarrow \epsilon$		

Ways of writng CFG

$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to id$$

E → E + E | E *E | (E) | id

E → E + E
| E * E
| (E)
| id

Context-free Grammars: Terminology

- The ***language*** of a grammar G = (N,T,P,S) is

$$L(G) = \{ w \mid w \in T^* \text{ and } S \Rightarrow^* w \}.$$

The language of a grammar contains only strings of terminal symbols.

Derivation

Derivation

(1)

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

- $E \xRightarrow{E \rightarrow E+E} E + E$

  $\xRightarrow{E \rightarrow id} id + E$

  $\xRightarrow{E \rightarrow id} id + id$

is a derivation of the terminal **string** *id + id* from E

- In a derivation, a production is applied at each step, **to replace a nonterminal** by the right-hand side of the corresponding production.
- The above derivation is represented in short as,

  $E \Rightarrow^{*} id + id,$

and is read as

E derives id + id

Derivations: Example

- Grammar for palindromes: G = (N, T, P, S),
  - N = {S},
  - T = {0, 1},
  - P = {

    $S \to 0\,S\,0$
    $\;\mid\; 1\,S\,1$
    $\;\mid\; 0$
    $\;\mid\; 1$
    $\;\mid\; \varepsilon$

    }.

(2)

$S \to 0S0$
$S \to 1S1$
$S \to 0$
$S \to 1$
$S \to \epsilon$

- A derivation of the string **10101**:

  $S \Rightarrow 1\,S\,1$ (using $S \to 1S1$)

  $\Rightarrow 1\,0S0\,1$ (using $S \to 0S0$)

  $\Rightarrow 10101$ (using $S \to 1$)

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example

S → aAS | a
A → SbA | SS | ba

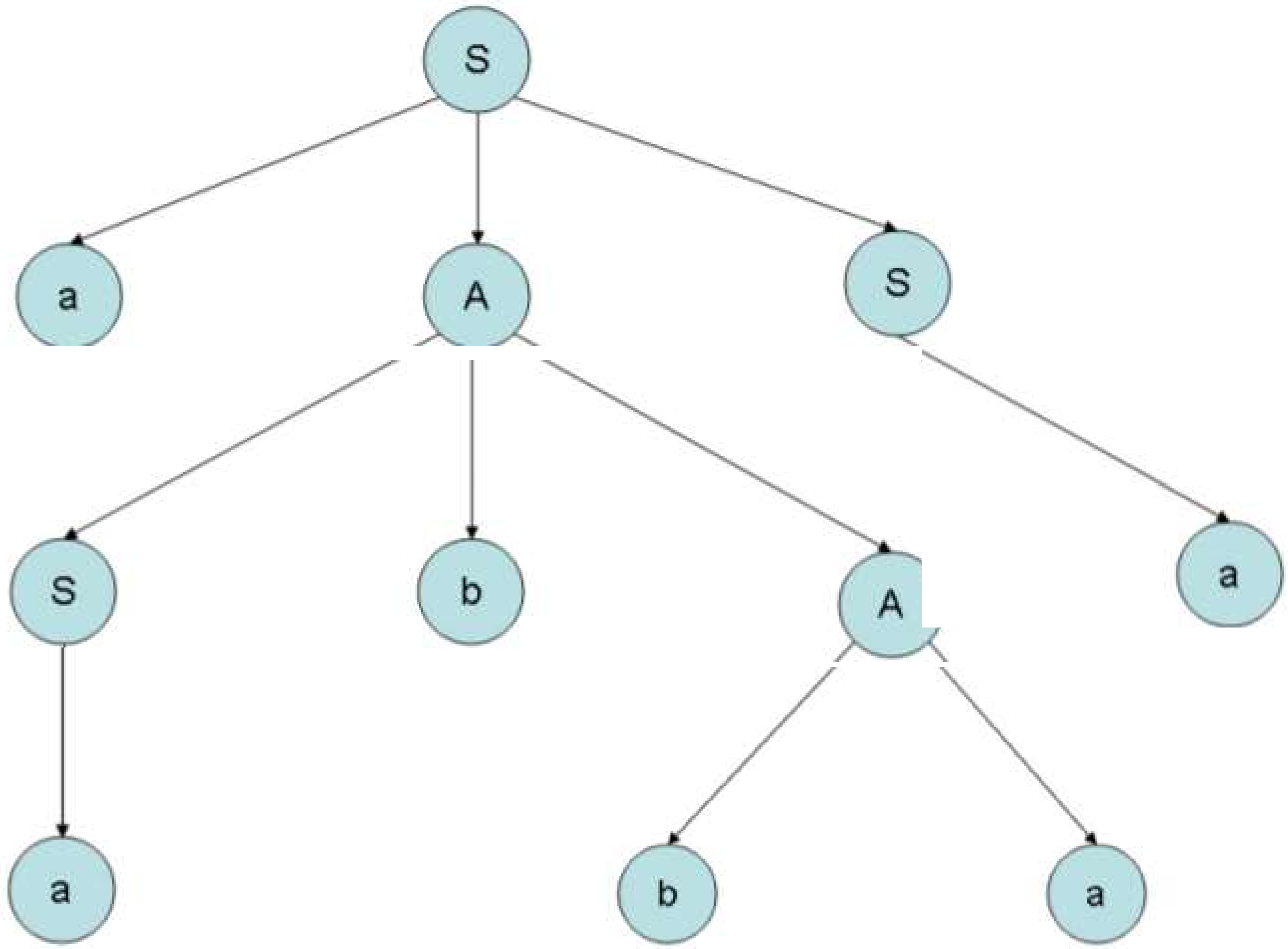
S => aAS

=> aSbAS

=> aabAS

=> aabbaS

=> aabbaa



Leftmost and Rightmost Derivations

- A *leftmost derivation* is one where, at each step, the leftmost nonterminal is replaced.

  (analogous for *rightmost derivation*)

- *Example*: a grammar for arithmetic expressions:

  $E \rightarrow E + E \mid E * E \mid \mathbf{id}$

Leftmost derivation:

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow \mathbf{id} + E * E$
$\Rightarrow \mathbf{id} + \mathbf{id} * E$
$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

Rightmost derivation:

$E \Rightarrow E + E$
$\Rightarrow E + E * E$
$\Rightarrow E + E * \mathbf{id}$
$\Rightarrow E + \mathbf{id} * \mathbf{id}$
$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$

Context-free Grammars: L(G)

A grammar for palindromic bit-strings:

G = (N, T, P, S), where:

- V = { S, B }
- T = {0, 1}
- P = { $S \rightarrow B$,

  $S \rightarrow \varepsilon$,

  $S \rightarrow 0\ S\ 0$,

  $S \rightarrow 1\ S\ 1$,

  $B \rightarrow 0$,

  $B \rightarrow 1$

  }

- $L(G) = \{ w \mid w \in T^* \text{ and } S \Rightarrow^* w \}$.

L(G) = { 0, 1, 00,11,101,010, 111, 000……. }

Ambiguous Grammar

Ambiguous Grammar

- If some word or string **w** in L(G) has two or more leftmost derivation or rightmost derivation parse trees, then G is said to be ambiguous
- The grammar,
  - E → E + E
    |E ∗ E
    |(E)
    |id is ambiguous,
- But the following grammar for the same language is **unambiguous**
  - E → E + T
    |T
  - T → T ∗ F
    |F
  - F → (E)|id

Ambiguity Example 1

$$
\begin{aligned}
E \rightarrow\ & E + E \\
|\ & E * E \\
|\ & (E) \\
|\ & id
\end{aligned}
$$

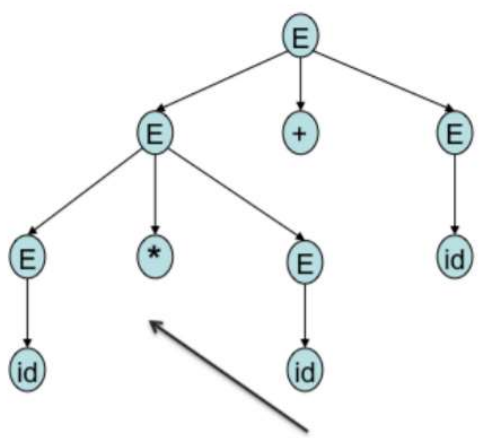
word = id * id + id

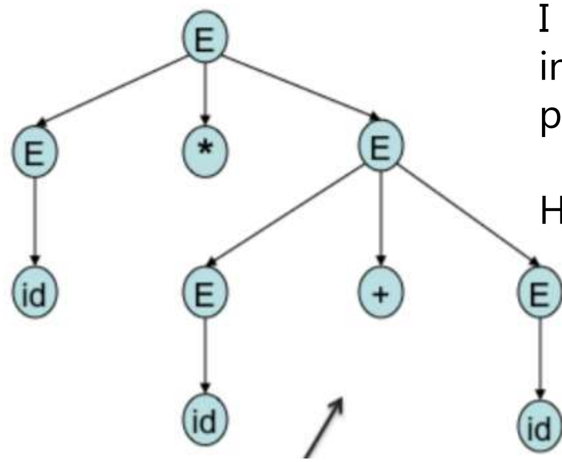
More than one left-most derivation

Two different leftmost derivation

E => E+E
=> E*E+E
=> id*E+E
=> id*id+E
=> id*id+id

E => E*E
=> id*E
=> id*E+E
=> id*id+E
=> id*id+id





For the same string using this grammar I am able to Generate two different derivation tree
OR

I am able to derive the string in two different ways by performing leftmost derivation.

Hence the grammar is **AMBIGOUS**

Dealing with Ambiguity

1. Transform the grammar to an equivalent unambiguous grammar.
2. Use *disambiguating rules* along with the ambiguous grammar to specify which parse to use.

Comment: It is not possible to determine algorithmically whether:

- Two given CFGs are equivalent;
- A given CFG is ambiguous.