Compiler Design

Dr. Sahar kamal

Syntax Analysis

Lecture 5

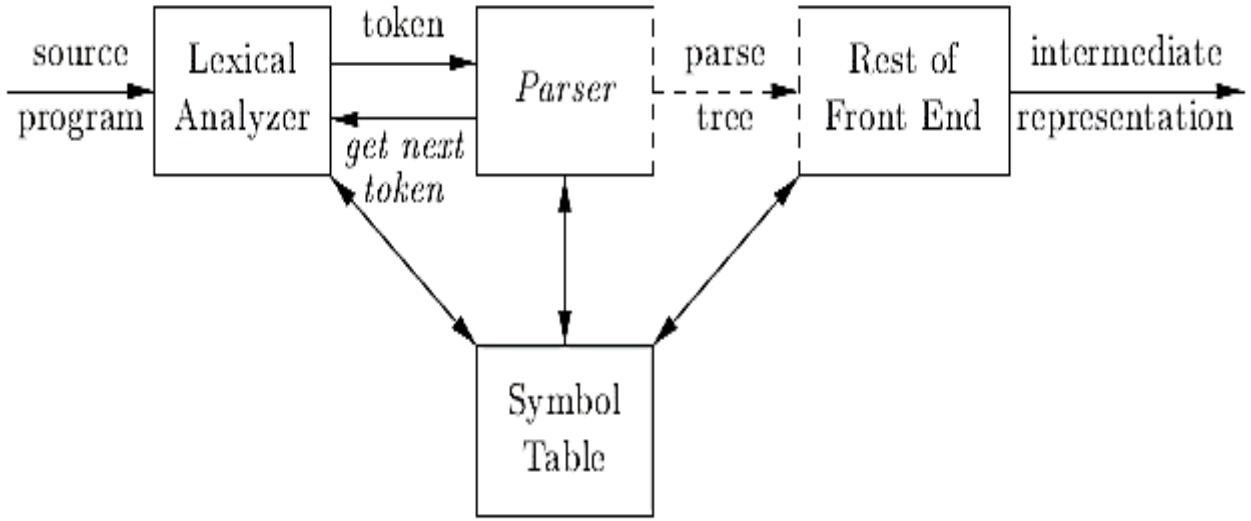
What is Syntax Analysis?

- The syntax analysis phase gets its input from the lexical analyzer which is a string of tokens. It verifies whether the provided string of tokens is grammatically correct or not.

- If that string of tokens is grammatically incorrect, it reports the syntax error.
  1. If that string of tokens is grammatically correct, it produces a parse tree for that string.
  2. Later the syntax analyzer forwards this parse tree to the next front end for processing. We also refer to the syntax analyzer as the parser

The Role of the Parser

- In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig, and verifies that the string of token name scan be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program.



Types of parser

- There are three general types of parsers for grammars:

1. Universal
2. top-down
3. bottom-up.

➢Universal parsing

Though universal parsing can parse any type of grammar. But it is quite ineffective to be used in the production compiler. So usually, we only use two methods for parsing top-down and bottom.

➢Top-down Parsing

In the top-down method, the parser builds the parse tree starting from the top. That means it starts from the root of the parse tree, traversing towards the bottom i.e. the leaves of the parse tree.

➢Bottom-up Parsing

In the bottom-up method, the parser builds the parse tree starting from the bottom. This implies it starts from the leaves of the parse tree, traversing upwards to the top i.e. root of the parse tree

Syntax Error Handling

Common programming errors can occur at many different levels.

- *Lexical* errors include misspellings of identifiers, keywords, or operators — e.g., the use of an identifier `elipseSize` instead of `ellipseSize` — and missing quotes around text intended as a string.

- *Syntactic* errors include misplaced semicolons or extra or missing braces; that is, "{" or "}." As another example, in C or Java, the appearance of a `case` statement without an enclosing `switch` is a syntactic error (however, this situation is usually allowed by the parser and caught later in the processing, as the compiler attempts to generate code).

- *Semantic* errors include type mismatches between operators and operands, e.g., the return of a value in a Java method with result type `void`.

- *Logical* errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==. The program containing = may be well formed; however, it may not reflect the programmer's intent.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect

Grammar

- A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

  if (expression) statement else statement

- That is, an if-else statement is the concatenation of the keyword if, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword else, and another statement. Using the variable expr to denote an expression and the variable stmt to denote a statement, this structuring rule can be expressed as

  Statement ::= if Expr then Stmt ElseClause
  ElseClause ::= else Stmt |ε

- in which the arrow may be read as “can have the form." Such a rule is called a production. In a production, lexical elements like the keyword if and the parentheses are called terminals. Variables like expr and Stmt represent sequences of terminals and are called non-terminals.

context-free grammar

A context-free grammar (CFG) has four components:

1. *Terminals* are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer. In (4.4), the terminals are the keywords **if** and **else** and the symbols "(" and ")."

2. *Nonterminals* are syntactic variables that denote sets of strings. In (4.4), *stmt* and *expr* are nonterminals. The sets of strings denoted by nonterminals help define the language generated by the grammar. Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

3. In a grammar, one nonterminal is distinguished as the *start symbol*, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.

context-free grammar

4. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production

- consists of:

1. A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.
2. The symbol !. Sometimes ::= has been used in place of the arrow.
3. A body or right side consisting of zero or more terminals and non terminals. The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Example 4.5 : The grammar in Fig. 4.2 defines simple arithmetic expressions. In this grammar, the terminal symbols are

id + - * / ( )

The nonterminal symbols are *expression*, *term* and *factor*, and *expression* is the start symbol □

$$
\begin{array}{rcl}
\mathit{expression} & \rightarrow & \mathit{expression} \texttt{ + } \mathit{term} \\
\mathit{expression} & \rightarrow & \mathit{expression} \texttt{ - } \mathit{term} \\
\mathit{expression} & \rightarrow & \mathit{term} \\
\mathit{term} & \rightarrow & \mathit{term} \texttt{ * } \mathit{factor} \\
\mathit{term} & \rightarrow & \mathit{term} \texttt{ / } \mathit{factor} \\
\mathit{term} & \rightarrow & \mathit{factor} \\
\mathit{factor} & \rightarrow & \texttt{(} \mathit{expression} \texttt{)} \\
\mathit{factor} & \rightarrow & \textbf{id}
\end{array}
$$

Figure 4.2: Grammar for simple arithmetic expressions

Grammar

- Grammar G=(N,T,P,S)
- N : a set of nonterminal symbols
- T : a set of terminal symbols, tokens
- P : a set of production rules
- S : a start symbol, S∈N
- EX: Grammar G for a language L={9-5+2, 3-1, ...}
- G=(N,T,P,S)
- N={list , digit}
- T={0,1,2,3,4,5,6,7,8,9,-,+}
- P : list -> list + digit
- list -> list - digit
- list -> digit
- digit -> 0|1|2|3|4|5|6|7|8|9
- S=list

Notational Conventions

To avoid always having to state that "these are the terminals," "these are the nonterminals," and so on, the following notational conventions for grammars will be used throughout the remainder of this book.

1. These symbols are terminals:
   (a) Lowercase letters early in the alphabet, such as a, b, c.
   (b) Operator symbols such as $+$, $*$, and so on.
   (c) Punctuation symbols such as parentheses, comma, and so on.
   (d) The digits $0, 1, \ldots, 9$.
   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.
2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.
   (d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by E, T, and F, respectively.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where A is the head and α the body.

6. A set of productions $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k$ with a common head A (call them *A-productions*), may be written $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Example 4.6: Using these conventions, the grammar of Example 4.5 can be rewritten concisely as

$$\begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T / F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

The notational conventions tell us that E, T, and F are nonterminals, with E the start symbol. The remaining symbols are terminals. $\square$

Example 1:

Construct the CFG for the language having any number of a's over the set ∑= {a}.

Solution:

As we know the regular expression for the above language is

RE = a*

Production rule for the Regular expression is as follows:

1.S → aS rule 1

2.S → ε rule 2

Example 2:

Construct a CFG for the regular expression (0+1)*

Solution:

The CFG can be given by,

Production rule (P):

1.S → 0S | 1S

2.S → ε

Derivations

- Beginning with the start symbol, each rewriting step replaces a nonterminal by the body of one of its productions. This derivational view corresponds to the top-down construction of a parse tree, but the precision afforded by derivations will be especially helpful when bottom-up parsing is discussed.
- For example, consider the following grammar, with a single nonterminal E,
- which adds a production E ->- E to the grammar :

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

$$E \Rightarrow -E$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Derivations

- To decide which non-terminal to be replaced with production rule, we can have two options.
- **Left-most Derivation**
  - If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.
- **Right-most Derivation**
  - If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example

Production rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: id + id * id

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$

Derive the string "aabbabba" for leftmost derivation and rightmost derivation using a CFG given by,

1.S → aB | bA

2. A → a | aS | bAA

3.B → b | aS | aBB|bS

Leftmost derivation:

1.S

2.aB S → aB

3.aaBB B → aBB

4.aabB B → b

5.aabbS B → bS

6.aabbaB S → aB

7.aabbabS B → bS

8.aabbabbA S → bA

9.aabbabba A → a

Rightmost derivation:

1.S

2.aB S → aB

3.aaBB B → aBB

4.aaBbS B → bS

5.aaBbbA S → bA

6.aaBbba A → a

7.aabSbba B → bS

8.aabbAbba S → bA

9.aabbabba A → a

Derive the string "00101" for leftmost derivation and rightmost derivation using a CFG given by,

1.S → A1B

2.A → 0A | ε

3.B → 0B | 1B | ε

Leftmost derivation:

1.S

2.A1B

3.0A1B

4.00A1B

5.001B

6.0010B

7.00101B

8.00101

Rightmost derivation:

1.S

2.A1B

3.A10B

4.A101B

5.A101

6.0A101

7.00A101

8.00101

Pars tree

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminal. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation. Ex: -(id + id)

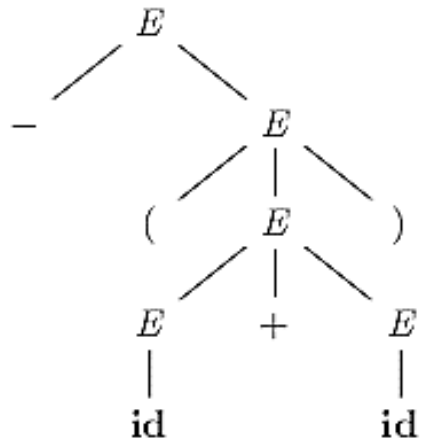


Figure 4.3: Parse tree for $-(\mathbf{id} + \mathbf{id})$

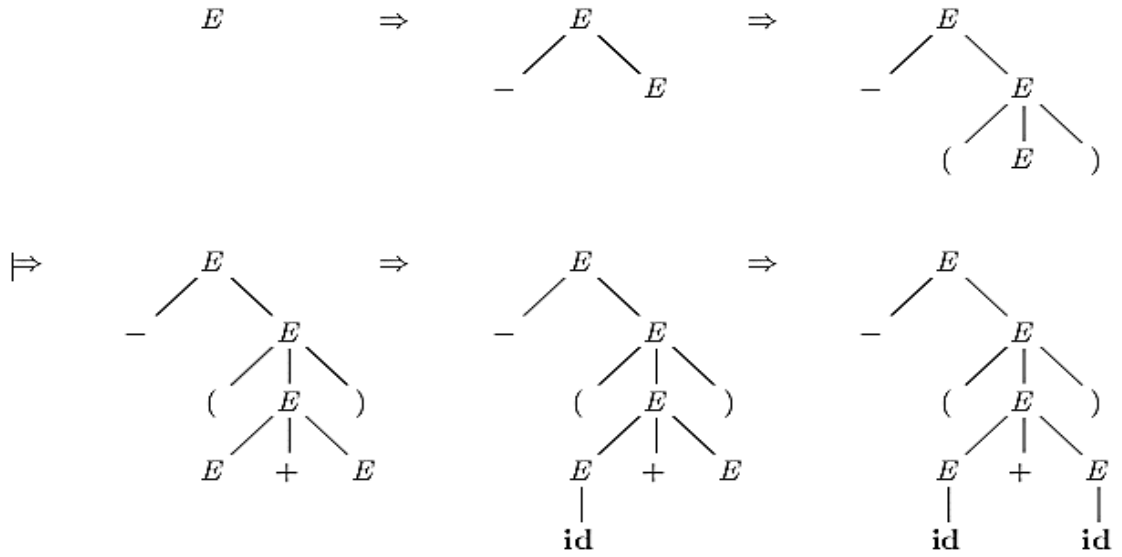


Figure 4.4: Sequence of parse trees for derivation (4.8)

parse tree

- **In a parse tree:**

1. All leaf nodes are terminals.
2. All interior nodes are non-terminals.
3. A parse tree depicts associativity and precedence of operators.

Ambiguity

- a grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence
- A grammar G is said to be ambiguous if it has more than one parse tree left or right derivation for at least one string.

Example 4.11: The arithmetic expression grammar (4.3) permits two distinct leftmost derivations for the sentence $\mathbf{id} + \mathbf{id} * \mathbf{id}$:

$$
\begin{array}{lcl}
E & \Rightarrow & E + E \\
 & \Rightarrow & \mathbf{id} + E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lcl}
E & \Rightarrow & E * E \\
 & \Rightarrow & E + E * E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
$$

Figure 4.5: Two parse trees for **id+id∗id**

The corresponding parse trees appear in Fig. 4.5.

Note that the parse tree of Fig. 4.5(a) reflects the commonly assumed precedence of + and ∗, while the tree of Fig. 4.5(b) does not. That is, it is customary to treat operator ∗ as having higher precedence than +, corresponding to the fact that we would normally evaluate an expression like $a + b * c$ as $a + (b * c)$, rather than as $(a + b) * c$. □

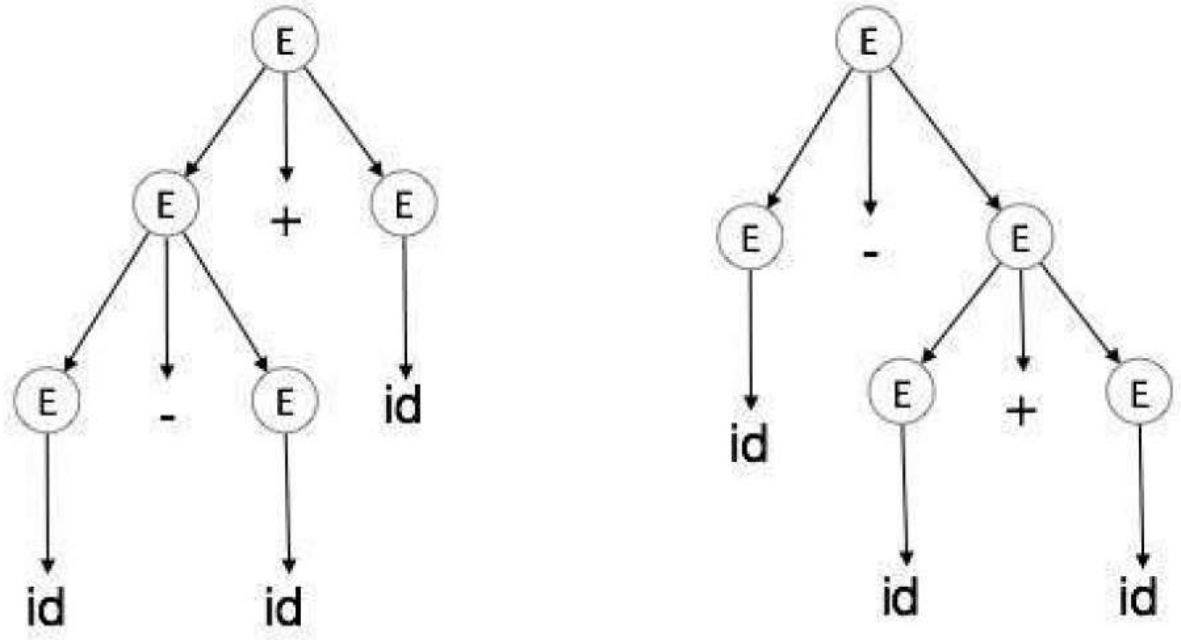
Example

E → E + E

E → E – E

E → id

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Example :

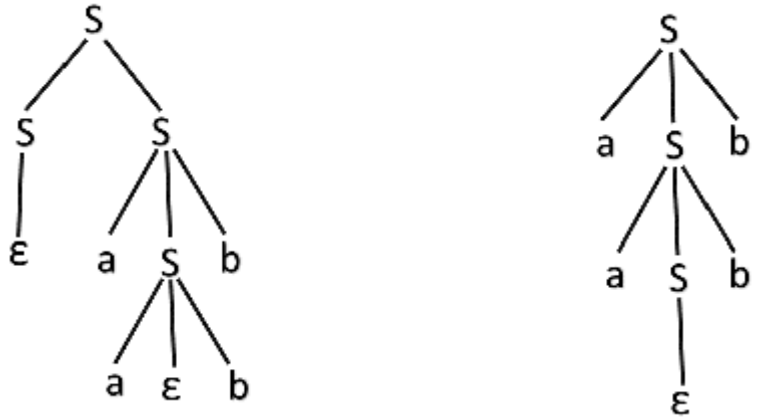
Check whether the given grammar G is ambiguous or not.

S → aSb | SS

S → ε

Solution:

For the string "aabb" the above grammar can generate two parse trees

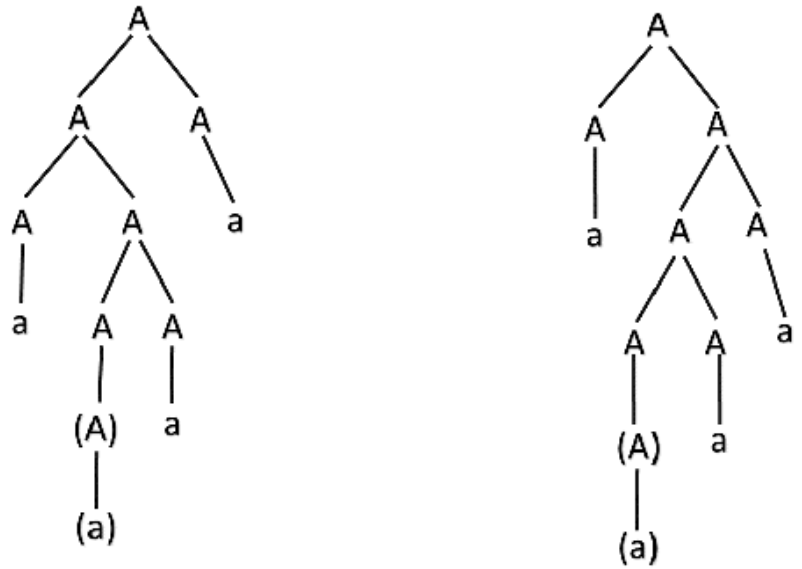


Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

Example :
Check whether the given grammar G is ambiguous or not.

1.A → AA
2.A → (A)
3.A → a

For the string "a(a)aa" the above grammar can generate two parse trees:



Since there are two parse trees for a single string "a(a)aa", the grammar G is ambiguous.

Types of Parsers

Types of Parser:
The parser is mainly classified into two categories, i.e., Top-down Parser, and Bottom-up Parser. These are explained below