# Syntax Analysis

## Role of Parsers

A parser for a grammar 'G' is a program that takes as input a string 'w' and produces as output either a parse tree for 'w' if 'w' is a sentence of G or an error message indicating that 'w' is not a sentence of G.

The parser has two functions:

- It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occurs in the patterns that are permitted by the specification for the source language.
- It also imposes on the tokens a tree like structure that is used by the subsequent phases of the compiler.

For example, if a PL/I program contains the expression

A+/B

Then after lexical analysis this expression might appear to the syntax analyzer as the token sequence:

id+/id

On seeing the / after +, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the rules of PL/I expression.

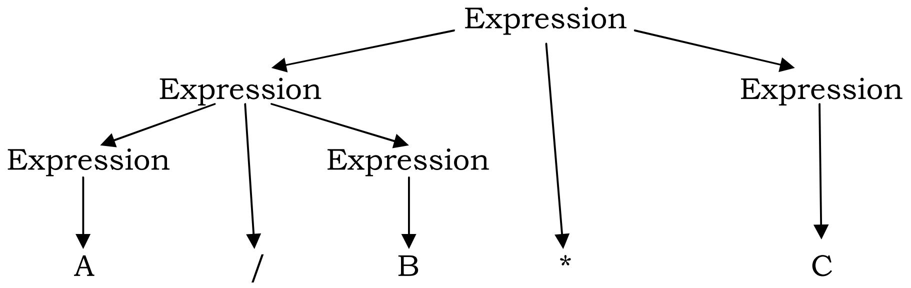
The second aspect of syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped together. For example, the following expression:

A/B*C

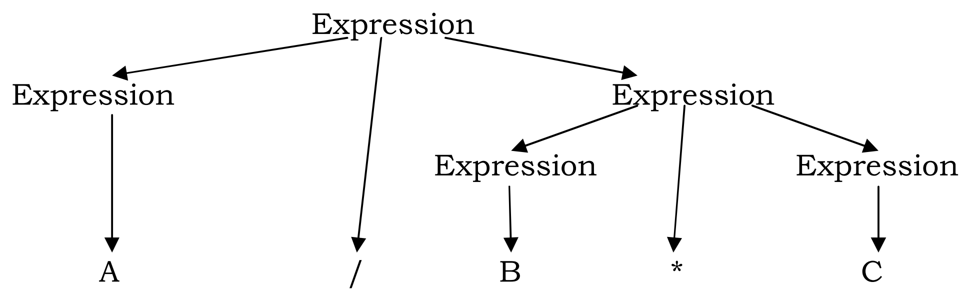
has two possible interpretations:

a) Divide A by B, and then multiply by C; or
b) Multiply B by C, and then use the result to divide A.

Each of these two interpretations can be represented in terms of a parse tree as shown:



(a)



(b)

The language specification must tell that which of the interpretations (a) and (b) is to be used, and in general, what hierarchical structure each source program has. These rules form the syntactic specification of a programming language.

## Types of Parsers

There are two types of parsers:

1. Bottom-up parser, and
2. Top-down parser.

### 1 Bottom-up Parser

These build the parsers from the bottom or leaves to the top i.e., root. It is also called as ***Shift-Reduce-Parser*** because it consists of shifting the input symbols onto a stack until the right side of a production appears on the top of the Stack.

The right side then be replaced or reduced by the symbol on the left side of the production. One kind of shift-reduce parsers are ***Operator-precedence parsers*** and more general type of shift-reduce parsers are ***LR parsers***.

### 2 Top-down Parser

These start with the root and work down to the leaves. This method is called as ***Recursive-Descent Parsing*** and its tabularized form is called as Predictive Parser and a special kind of predictive parser is called as ***LL Parser***.

## Ambiguous Grammar

A grammar that produces more than one parse tree for same sentence is said to be ***ambiguous***. An ambiguous grammar is one that produces more than one left most or more than one right most derivations for same sentence. For some type of parsers, it is desirable that the grammar be made unambiguous, for if it is not, then it is difficult to uniquely determine which parse tree to select for a sentence.

For example, consider the following grammar for arithmetic expressions involving +, -, *, /, and ↑ (exponentiation).

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid ( E ) \mid -E \mid id$

This grammar is ambiguous.

However, this grammar can be disambiguated by specifying the associativity and precedence of the arithmetic operators. Suppose the operators have the following precedence in decreasing order:

| | |
|---|---|
| - | (unary minus) |
| ↑ | |
| * | / |
| + | - |

Suppose further ↑ operator be right associative [e.g. a ↑ b ↑ c is to mean a ↑ (b ↑ c)] and other binary operators to be left associative [e.g. a – b – c is to mean (a - b) - c]. These precedence and associativities are the ones customarily used in mathematics and in many, but not all, programming languages. These rules concerning the associativity and precedence of operators are sufficient to disambiguate such grammars. For each sentence of these grammars, there is exactly one parse tree that groups operands of operators according to these associativity and precedence rules.

Thus using these precedence and associativity rules the above given ambiguous grammar can be simplified as follows:

| | |
|---|---|
| Step I: | $E \rightarrow E \mid id$ |
| II: | $E \rightarrow - E \mid id$ |
| III: | $E \rightarrow - E \uparrow E \mid id$ |
| IV: | $E \rightarrow - E \uparrow E * E \mid id$ |
| V: | $E \rightarrow - E \uparrow E * E / E \mid id$ |
| VI: | $E \rightarrow - E \uparrow E * E / E + E \mid id$ |
| VII: | $E \rightarrow - E \uparrow E * E / E + E – E \mid id$ |

## Context Free Grammar

A language is said to be ***context-free*** if the productions may be applied repeatedly to expand the non-terminals in a string of non-terminals and terminals. A grammar that supports the context-free languages is called as a ***context-free grammar (CFG)***.

For example, consider the following grammar for an arithmetic expression:

$E \rightarrow E + E \mid E * E \mid ( E ) \mid - E \mid id$ --(1)

In general, a grammar involves four quantities:

- Terminals
- Non-terminals
- Start symbol, and
- Productions

**Terminals:**

Terminals are the basic symbols of which strings in the language are composed. The term "token name" is a synonym for “Terminal". For e.g., +, id, *, (, ), a, b, etc.

**Non-terminals:**

Non-terminals are the special symbols or syntactic variables that denote the set of strings. The sets of strings denoted by non-terminals help define the language generated by the grammar. For e.g., E, F, T in the productions as follows:

$E \rightarrow F + T$
$F \rightarrow T * E$
$T \rightarrow ( E ) / id$

**Productions:**

The productions of a grammar defines the ways in which the syntactic categories may be built up from one another and form the terminals. Each production consists of a non-terminal followed by an arrow, followed by a string of terminals and non-terminals as shown above.

**Start Symbol:**

One non-terminal is selected as a ***start symbol*** and it denotes the language concerned.

## Notational Conventions

To avoid always having to state that “these are non-terminals”, or “these are terminals”, and so on, some notational shorthands shall be employed. These may be summarized as follows:

1) Following symbols are usually non-terminals:
    a. Lower-case italic names such as *expression, statement, operator, etc.*
    b. Upper-case letters near the beginning of the alphabet, such as A, B, C.
    c. The letter, S, which, when appears, is usually the start symbol.

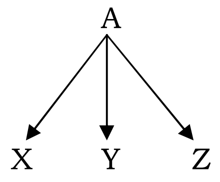
2) Following symbols are usually terminals:
    a. Lower-case letters such as a, b, c.
    b. Operator symbols such as +, -, etc.
    c. Punctuation symbols such as parentheses, comma, etc.
    d. The digits 0, 1, …., 9.
    e. Bold-face strings such as **id** or **if**.

3) Capital symbols near the end of the alphabet, chiefly X, Y, Z, represents grammar symbols, i.e., either non-terminals or terminals.
4) Small letters near the end of the alphabet, chiefly u, v, z, represents the strings of terminals.
5) Lower case Greek letters, $\alpha$, $\beta$, $\gamma$, for example, represents strings of grammar symbols. Thus a generic production could appear as $A \rightarrow \alpha$, indicating that there is a single non-terminal A on the left side of the production and a string of grammar symbols '$\alpha$' to the right side of the production.
6) If $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, … , $A \rightarrow \alpha_k$ are all productions with A on the left, it can be written as $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots. \mid \alpha_k$. Here $\alpha_1$, $\alpha_2$, …. , $\alpha_k$ are the alternates of A.
7) Unless otherwise stated, the left side of the first production is the start symbol.

## Parse Trees and Derivations

A graphical representation for derivations that filters out the choice regarding the replacement order is called the **PARSE TREE**. It has the important purpose of making explicit the hierarchical syntactic structure of sentences that is implied by the grammar.

Each interior node of the parse tree is labeled by some non-terminal A, and the children of the node are labeled, from left to right by the symbols in the right side of the production by which this A can be replaced in the derivation. For e.g., if $A \rightarrow XYZ$ is a production used at some step of derivation, then the parse tree for that derivation will have the following sub tree:



The leaves of the parse tree are labeled by the non-terminals or terminals, and read from left to right. They constitute a sentential form called the ***YIELD*** or ***FRONTIER*** of the tree.

For e.g., the parse tree for – (id + id) implied by the derivation is shown as follows:

$$\begin{tikzcd}
 & E \arrow[dl] \arrow[dr] & & \\
- & & E \arrow[dl] \arrow[d] \arrow[dr] & \\
 & ( & E \arrow[dl] \arrow[d] \arrow[dr] & ) \\
 & E \arrow[d] & + & E \arrow[d] \\
 & Id & & id
\end{tikzcd}$$

<u>Parse Tree</u>

The parse tree ignores variations in the order in which symbols are replaced. These variations in the order in which productions are applied can also be eliminated by considering only leftmost (or rightmost) derivations. It is not hard to see that every parse tree has associated with it a unique leftmost and a unique rightmost derivation. However, it should not be assumed that every sentence necessarily has only one parse tree or only one leftmost or rightmost derivation.