Syntax Analysis (Parsing)

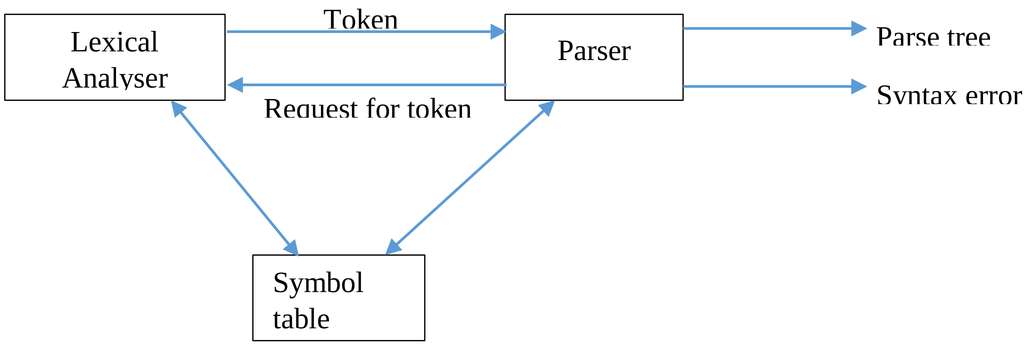
This is the process of analysing a sequence of tokens to determine their grammatical structure with respect to a given formal grammar.

It is the most important phase of a compiler. A syntax analyser (**parser**) checks for the correct syntax and builds a data structure (**parse tree**) implicit in the input tokens i.e. it considers the sequence of tokens for possible valid constructs of the programming language.

Role of a Parser

1. To identify language constructs present in a given input program. If the parser determines input to be valid, it outputs the presentation of the input in form of a parse tree.
2. If the input is grammatically incorrect, the parser declares the detection of syntax error in the input. In this case, a parse tree is not produced.

Illustration



Error Handling

Error handling is one of the most important features of any modern compiler. The challenge in error handling is to have a good guess of possible mistakes a programmer can do and come up with strategies to point these errors to the user in a very clear and unambiguous manner.

Common errors occurring in a program can be classified into the following:

a. **Lexical errors** – these are mainly spelling mistakes and accidental insertion of foreign characters. They are detected by the lexical analyser.
b. **Syntax errors** – these are grammatical mistakes such as unbalanced parenthesis in an expression, ill-formed constructs. They are the most common types of errors in a program and are detected by the parser.
c. **Semantic errors** – are errors due to undefined variables, incompatible operands to an operator etc. they are detected by introducing some extra checks during parsing.
d. **Logical errors** – are errors such as infinite loops. There is no automatic way of detecting them; however, use of debugging tools may help the programmer identify them.

Generally, a good error handler should:

- Report errors accurately and clearly
- Recover from an error quickly
- Not slow down compilation of valid code

Good error handling is not easy to achieve

Error Recovery

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code, namely:

i). Panic mode
ii). Statement mode
iii). Error Productions
iv). Global Correction

Not all are supported by all parser generators

i). **Panic Mode: -** When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

ii). **Statement Mode: -** When a parser encounters an error, it tries to take corrective measures so that the rest of the inputs of the statement allow the parser to parse ahead. E.g. inserting a missing semicolon, replacing comma with a semicolon, etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

iii). **Error Productions: -** Some common errors that may occur in the code are known to the compiler designers. This requires the designers to create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered. The idea is to specify in the grammar known common mistakes, which essentially promotes common errors to alternative

Example:

Write 5 x instead of 5 * x

Add the production E → … | E E

iv). **Global Correction: -** The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Grammar

A grammar G is defined as a four-tuple with $< V_N, V_T, P, S >$ where:

- $V_{N:}$-this is the set of non-terminal symbols used to write the grammar.
- V_T: - this is the set of terminals (set of words of the language, lexicon or dictionary of words).
- P: - this is the set of production rules. It defines how a sequence of terminal and non-terminal symbols can be replaced by some other sequence.
- S: - $S \in V_N$ Is a special non-terminal called the start symbol of the grammar.

The language of the grammar $G =< V_N, V_T, P, S >$ denoted by $L(G)$ is defined as all those strings over V_T that can be generated by starting with the start symbol S then applying the production rules in P until no more non-terminal symbols are present.

Example

Consider the grammar to generate arithmetic expressions consisting of numbers and operator symbols i.e. +,-,*, /. Rules of the grammar can be written as follows:

$$\begin{aligned} &E \to EAE \\ &E \to (E) \\ &E \to number \\ &A \to + \\ &A \to - \\ &A \to * \\ &A \to / \end{aligned}$$

We can apply these rules to derive the expression $2 * (3 + 5 * 4)$ as follows

$E \to EAE \to EA(E) \to EA(EAE) \to EA(EAEAE) \to EA(EAEA4) \to EA(EAE * 4) \to EA(EA5 * 4) \to EA(E + 5 * 4) \to EA(3 + 5 * 4) \to E * (3 + 5 * 4) \to 2 * (3 + 5 * 4)$

In the grammar, E and A are non-terminals while the rest are terminals.

Context Free Grammar (CFG)
These are grammars that define context free languages and consist of production rules in which the left hand side contains only a single non-terminal and no terminals; the right hand side consists of either terminals, non-terminals or both. Notice that most programming language constructs belong to context free languages.

Examples

1. Write CFG for the language defined by the regular expression a : $A \rightarrow a$
2. Write CFG for the language defined by the regular expression a^*
3. Write CFG for the language defined by the regular expression *a(a)**

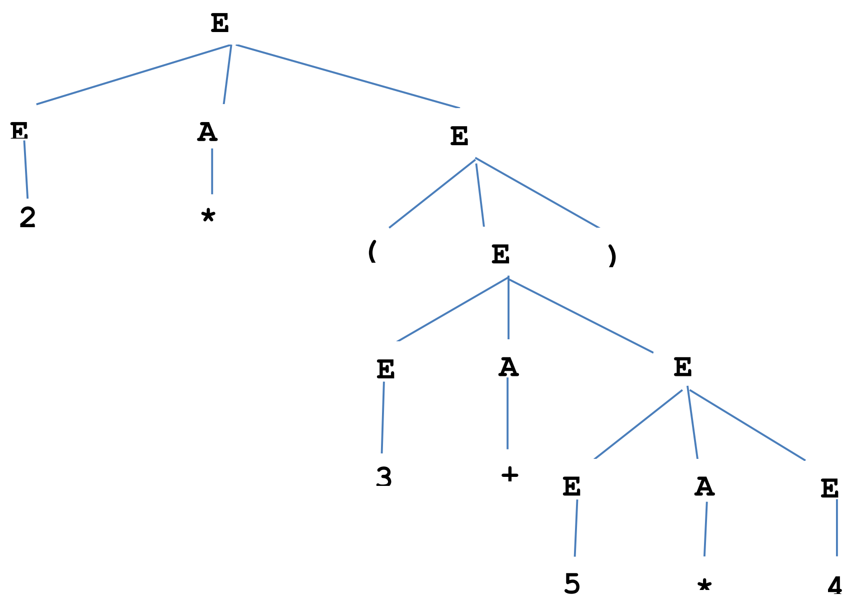
Notational Conventions
The following is the ***notational convention*** that would be used when defining grammars:

1. The following will be taken as terminals:
   - all operator symbols
   - punctuation symbols including parenthesis '()'
   - digits
   - lower case letters of the alphabet such as a, b, c
   - Lexemes such as id, number, while, etc.
2. The following will represent non-terminals:
   - Upper case latters of the alphabet such as A, B, C.
   - The letter S will represent the start symbol
   - Lowercase names such as expr, stmt etc.
3. A set of productions with the same left hand side like $A \rightarrow \alpha_1, A \rightarrow \alpha_2 \ldots A \rightarrow \alpha_n$ will be written as $A \rightarrow \alpha_1|\alpha_2| \ldots |\alpha_n$.
4. If no start symbol is represented, then the non-terminal appearing in the left hand side of the production rule will be taken as the start symbol.

Derivation
This is the process of generating a sequence of intermediary strings to expand the start symbol of the grammar to the desired string of terminals.

Each step of the derivation modifies an intermediary string to a new one by replacing a substring of it that matches the left hand side of the production rule by a string on the right hand side of the rule. The derivation is often represented by a parse tree.

Example
The following is the derivation of the string $2 * (3 + 5 * 4)$



Types of Derivation

There are two types of derivation

i). Left-most derivation
ii). Right-most derivation

Left-Most Derivation

Left-most derivation is a derivation in which the left-most non-terminal is replaced at each step. The intermediate strings are called ***left-sentential forms*** and consists of grammar symbols both terminals and non-terminals.

Right-Most Derivation

Right-most derivation is a derivation in which the right-most non-terminal is replaced at each step. The intermediary strings are called ***right-sentential forms***. Right-most derivation is often referred to as ***canonical representation***.

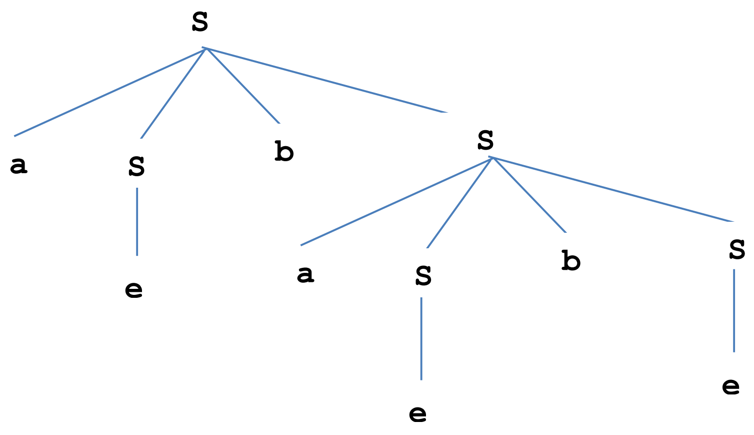
Ambiguity

A grammar is said to be **ambiguous** if there exists more than one parse tree for the same sentence e.g. consider the following grammar:

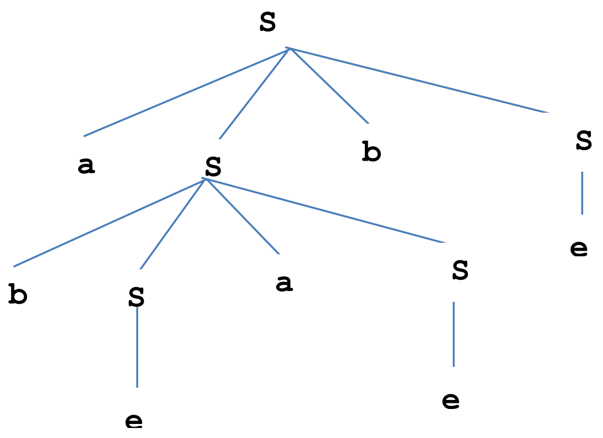
$$S \rightarrow aSbS|bSaS|e$$

Show that the grammar is ambiguous by giving two different parse trees for the string "$abab$".

First parse tree



Second parse tree



Dangling Else Ambiguity

Most programing languages have both if… then and if… then … else versions of the statement. The grammar rules are as follows:

```
stmt → if condition then stmt else stmt
     | if condition then stmt
```

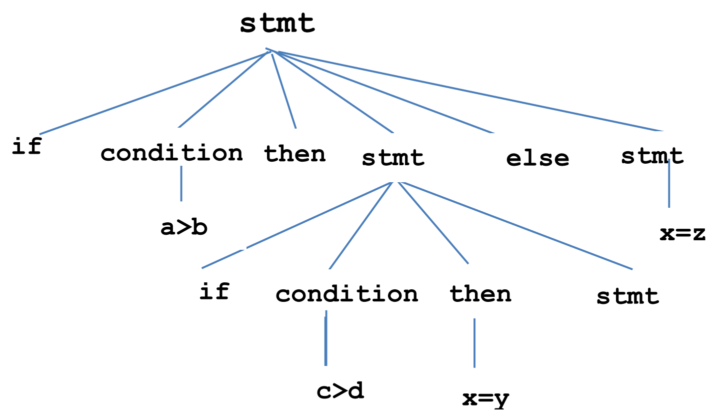
Consider the following code segment

$if\ a > b\ then$

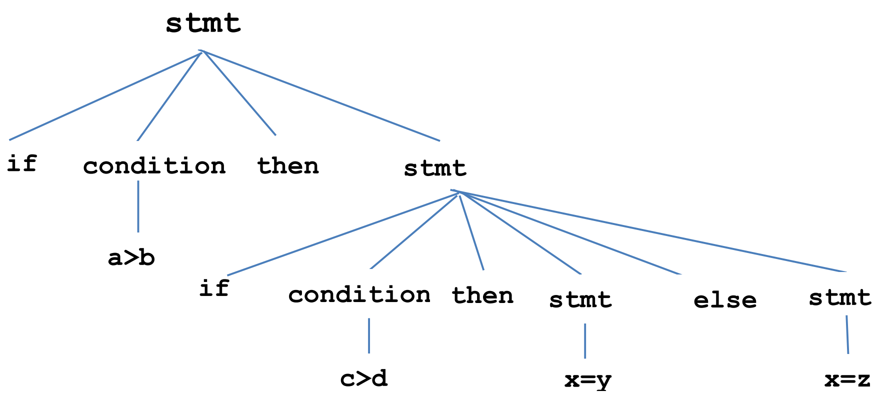
$\quad if\ c > d\ then\ x = y$

$\quad else\ x = z$

Two parse trees can be generated by the grammar as shown below:



i.e. the else is taken with the outer if statement



i.e. the else is taken with the inner if statement.

Notice that most programming languages accept the second one as the correct syntax.

Eliminating Ambiguity

Ambiguity may be eliminated by rewriting the grammar

e.g. if… then… else may be rewritten as follows:

$stmt \rightarrow matched\_stmt | unmatched\_stmt$

$matched\_stmt \rightarrow if\ condition\ then\ matched\_stmt\ else\ matched\_stmt$

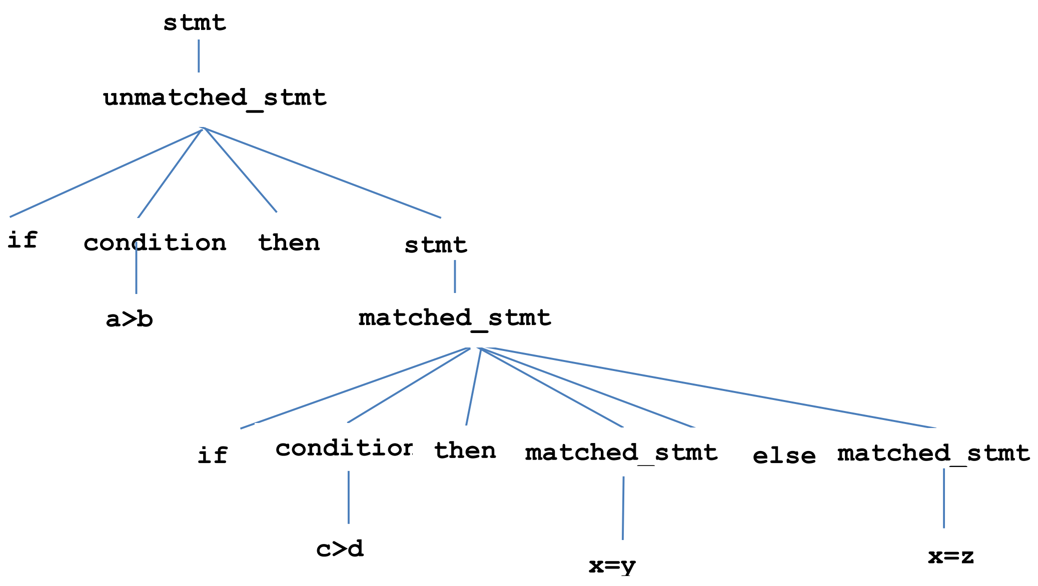
$\qquad |other\_stmt$

$unmatched\_stmt \rightarrow if\ condition\ then\ stmt$

$\qquad | if\ condition\ then\ matched\_stmt\ else\ unmatched\_stmt$

Other_stmt represents other statements apart from ‘if’.

The parse tree produced for the code segment is as follows:



Another technique to resolve ambiguity is to modify the grammar e.g. many programming languages require that “if” should have a matching “end if” as shown below:

$$stmt \rightarrow if\ condition\ then\ stmt\ else\ stmt\ endif$$

$$|\ if\ condition\ then\ stmt\ endif$$

Left-Recursion

A production is left-recursive if the left-most symbol on the right side is same as the non-terminal on the left side e.g.

$$A \rightarrow A \propto$$

There are two types of left-recursion

i). Immediate left-recursion
ii). General left-recursion

I). Immediate left-recursion

This occurs when a non-terminal A has a production rule of the form
$A \rightarrow A \propto |\beta$.

The immediate left-recursion can be eliminated by introducing a new non-terminal symbol for instance A' e.g. the production

$A \rightarrow A\alpha|\beta$

can be modified as follows:

$A \rightarrow \beta A'$

$\mathrm{A}' \rightarrow \alpha\mathrm{A}'|\mathrm{e}$

Therefore, the rule

$$A \rightarrow A\alpha_1|A\alpha_2| \ldots |A\alpha_m|B_1|B_2| \ldots |B_n$$

can be modified as

$A \rightarrow B_1A'|B_2A'| \dots |B_nA'$
$A' \rightarrow \alpha_1A'|\alpha_2A'| \dots |\alpha_mA'|e$

Example
Consider the following left recursive grammar for arithmetic expression;

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

Eliminate the left-recursion from the grammar

Solution

$E \rightarrow TE'$
$E' \rightarrow +TE'|e$
$T \rightarrow FT'$
$T' \rightarrow * FT'|e$
$F \rightarrow (E)|id$

II). General left-recursion
This is a left-recursion due to a number of production rules

e.g. Consider the grammar:

$S \rightarrow Aa$
$A \rightarrow Sb|c$

S Is left recursive since

$S \rightarrow Aa \rightarrow Sba$

General left-recursion is eliminated by the following algorithm:

Algorithm to eliminate left-recursion:

```
1. Arrange non-terminals in some order say $A_1, A_2, \dots, A_m$
2. $For\ i = 1\ to m,\ do$
     $For\ j = 1\ to i - 1\ do$
             $For\ each\ set\ of\ production$
                   $A_i \rightarrow A_j\gamma\ and\ A_j \rightarrow \delta_1|\delta_2| \dots |\delta_k$
                   $replace\ A_i \rightarrow A_j\gamma\ by\ A_i \rightarrow \delta_1\gamma|\delta_2\gamma| \dots |\delta_k\gamma$
3. $Eliminate\ immediate\ left\ recursion\ from\ all\ productions$
```

E.g. the grammar

$S \rightarrow Aa$
A $\rightarrow Sb|c$

Let the order of non-terminals be S, A.

For $i = 1$, the rule $S \rightarrow Aa$ remains since there is no immediate left-recursion.

For $i = 2$, $A \rightarrow Sb|c$ is modified as $A \rightarrow Aab|c$ which has immediate left recursion and once eliminated, we get:

$A \rightarrow cA'$
$A' \rightarrow abA'|e$

Left-factorization
If two productions for the same nonterminal begin with the same sequence of symbols, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

E.g.
If a top-down parser encounters a production like

$A \Longrightarrow \alpha\beta \,/\, \alpha\gamma \,/\, \ldots$

Then it cannot determine which production to follow to parse the string, as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions. Generally, rewrite the grammar in such a way that the overlapping productions are made into a single production that contains the common prefix of the productions and uses a new auxiliary nonterminal for the different suffixes.

Method
If $\alpha \neq \varepsilon$ then replace all of the A productions

$A \Rightarrow \alpha\beta1 \,/\, \alpha\beta2/ \ldots / \,\alpha\beta n$

with

$A \Rightarrow \alpha A'$

$A' \Rightarrow \beta1 \,/\, \beta2/ \ldots / \,\beta n$

where $A'$is a new non-terminal.

Repeat until no two alternatives for a single nonterminal have a common prefix.

Example

The above productions can be written as

$A \Rightarrow \alpha A'$
$A' \Rightarrow \beta \,/\, \gamma \,/\, \ldots$

Now the parser has only one production per prefix which makes it easier to take decisions.