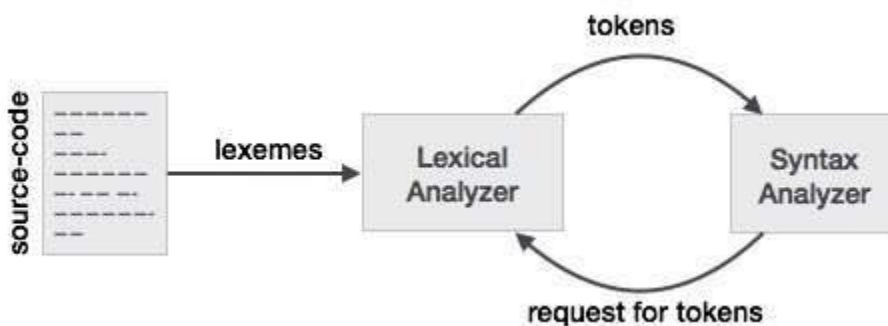


Unit-2

Compiler Design - Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ;  
(symbol) .
```

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols $\{0,1\}$ is a set of binary alphabets, $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$ is a set of Hexadecimal alphabets, $\{a-z, A-Z\}$ is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorialspoint is 14 and is denoted by $|\text{tutorialspoint}| = 14$. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ϵ (epsilon).

Special Symbols

A typical high-level language contains the following symbols:-

Arithmetic Symbols	Addition(+), Subtraction(-), Modulo(%), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.), Arrow(->)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, , , !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

```
int intvalue;
```

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

Regular Expressions

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as

$$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$$

- Concatenation of two languages L and M is written as

$$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$$

- The Kleene Closure of a language L is written as

L^* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$, then

- **Union** : $(r)|(s)$ is a regular expression denoting $L(r) \cup L(s)$
- **Concatenation** : $(r)(s)$ is a regular expression denoting $L(r)L(s)$
- **Kleene closure** : $(r)^*$ is a regular expression denoting $(L(r))^*$
- (r) is a regular expression denoting $L(r)$

Precedence and Associativity

- $*$, concatenation $(.)$, and $|$ (pipe sign) are left associative
- $*$ has the highest precedence
- Concatenation $(.)$ has the second highest precedence.
- $|$ (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x^* means zero or more occurrence of x.

i.e., it can generate $\{ \epsilon, x, xx, xxx, xxxx, \dots \}$

- x^+ means one or more occurrence of x.

i.e., it can generate $\{ x, xx, xxx, xxxx \dots \}$ or $x.x^*$

- $x?$ means at most one occurrence of x

i.e., it can generate either $\{x\}$ or $\{\epsilon\}$.

$[a-z]$ is all lower-case alphabets of English language.

$[A-Z]$ is all upper-case alphabets of English language.

$[0-9]$ is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = $[a - z]$ or $[A - Z]$

digit = $0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$ or $[0-9]$

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = (sign)?(digit)⁺

Identifier = (letter)(letter | digit)*

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Finite Automata

Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata is a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q₀)
- Set of final states (q_f)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), $Q \times \Sigma \rightarrow Q$

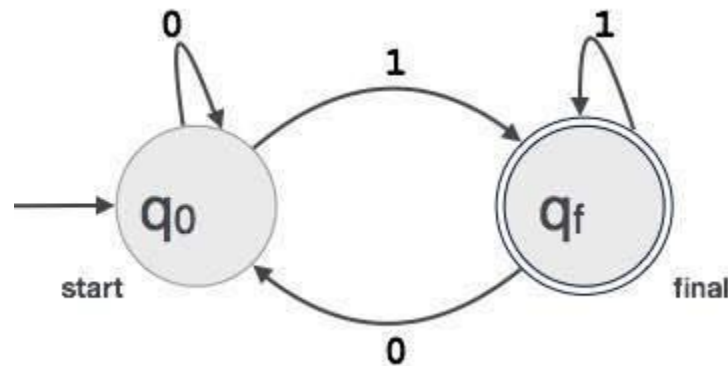
Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

- **States** : States of FA are represented by circles. State names are written inside circles.
- **Start state** : The state from where the automata starts, is known as the start state. Start state has an arrow pointed towards it.
- **Intermediate states** : All intermediate states have at least two arrows; one pointing to and another pointing out from them.
- **Final state** : If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows are one greater than even, i.e. **odd = even+1**.

- **Transition** : The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state. If automata stays on the same state, an arrow pointing from a state to itself is drawn.

Example : We assume FA accepts any three digit binary value ending in digit 1. $FA = \{Q(q_0, q_f), \Sigma(0,1), q_0, q_f, \delta\}$



Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:



It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- A set of **productions** (P). 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 a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on-terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{ w \mid w = w^R \}$ is not a regular language. But it can be described by means of CFG, as illustrated below:

$$G = (V, \Sigma, P, S)$$

Where:

$$V = \{ Q, Z, N \}$$

$$\Sigma = \{ 0, 1 \}$$

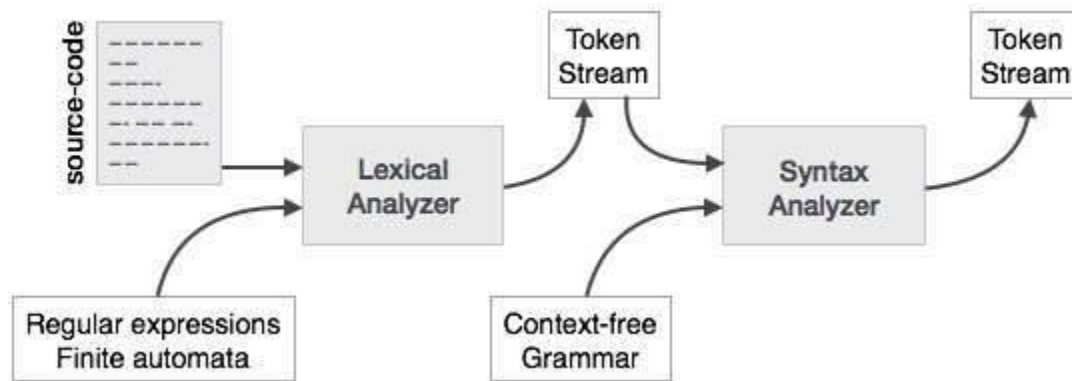
$$P = \{ Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \epsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1 \}$$

$$S = \{ Q \}$$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

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 → E + E
E → E * E
E → id
```

Input string: id + id * id

The left-most derivation is:

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

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

The right-most derivation is:

```
E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id
```

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

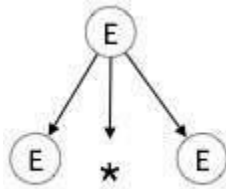
We take the left-most derivation of $a + b * c$

The left-most derivation is:

$E \rightarrow E * E$
 $E \rightarrow E + E * E$
 $E \rightarrow id + E * E$
 $E \rightarrow id + id * E$
 $E \rightarrow id + id * id$

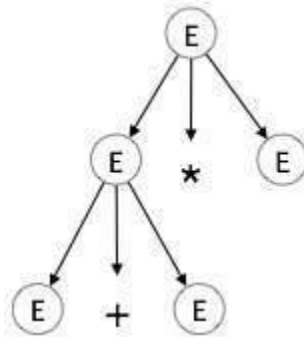
Step 1:

$E \rightarrow E * E$



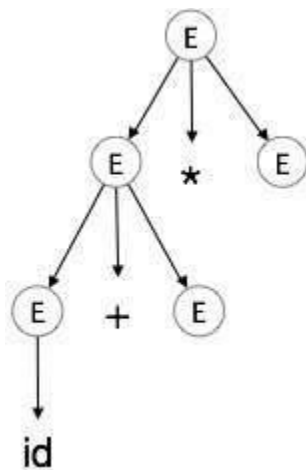
Step 2:

$E \rightarrow E + E * E$



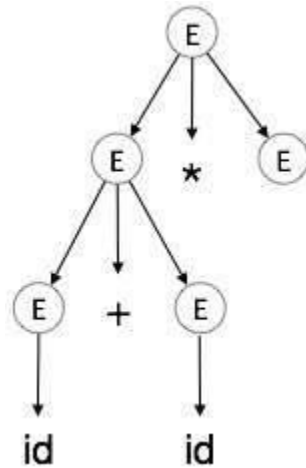
Step 3:

$E \rightarrow id + E * E$



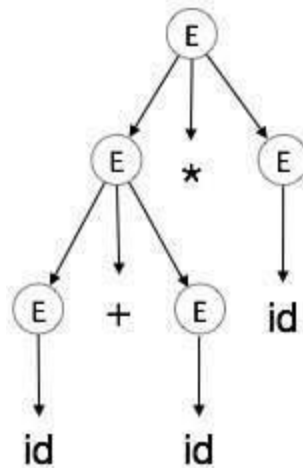
Step 4:

$E \rightarrow id + id * E$



Step 5:

$E \rightarrow id + id * id$



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

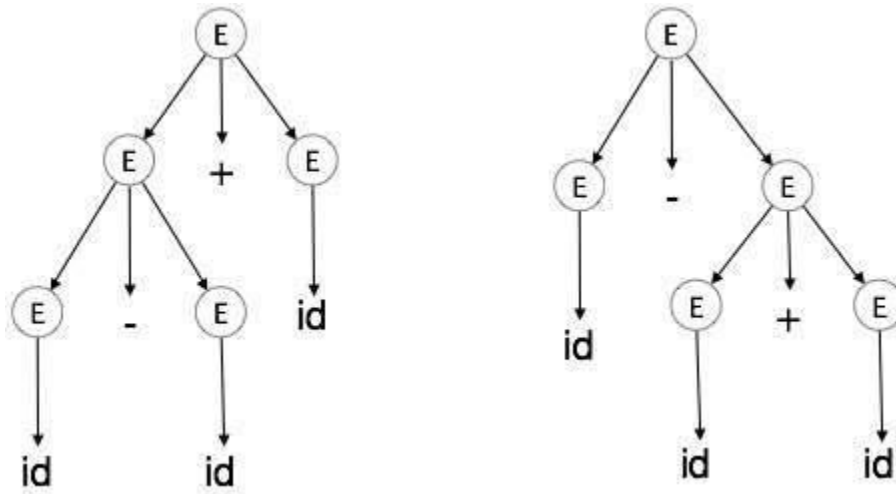
Ambiguity

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

$E \rightarrow E + E$
 $E \rightarrow E - E$
 $E \rightarrow id$

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.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

$id \text{ op } id \text{ op } id$

it will be evaluated as:

$(id \text{ op } id) \text{ op } id$

For example, $(id + id) + id$

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

$id\ op\ (id\ op\ id)$

For example, $id\ ^\wedge\ (id\ ^\wedge\ id)$

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, $2+3*4$ can have two different parse trees, one corresponding to $(2+3)*4$ and another corresponding to $2+(3*4)$. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically $*$ (multiplication) has precedence over $+$ (addition), so the expression $2+3*4$ will always be interpreted as:

$2\ +\ (3\ *\ 4)$

These methods decrease the chances of ambiguity in a language or its grammar.

Left Recursion

A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers. Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

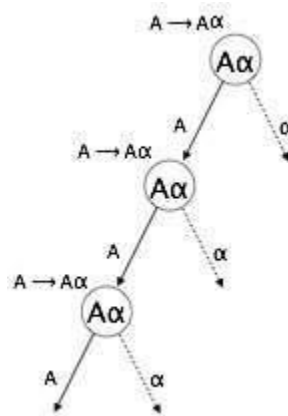
Example:

(1) $A \Rightarrow A\alpha \mid \beta$

(2) $S \Rightarrow A\alpha \mid \beta$
 $A \Rightarrow Sd$

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse the A , which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

Removal of Left Recursion

One way to remove left recursion is to use the following technique:

The production

$$A \Rightarrow A\alpha \mid \beta$$

is converted into following productions

$$\begin{aligned} A &\Rightarrow \beta A' \\ A' &\Rightarrow \alpha A' \mid \epsilon \end{aligned}$$

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

START

Arrange non-terminals in some order like $A_1, A_2, A_3, \dots, A_n$

```

for each i from 1 to n
{
  for each j from 1 to i-1
  {
    replace each production of form  $A_i \Rightarrow A_j \gamma$ 
    with  $A_i \Rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \delta_3 \gamma \mid \dots \mid \gamma$ 
    where  $A_j \Rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_n$  are current  $A_j$  productions
  }
}
eliminate immediate left-recursion

```

END

Example

The production set

$$\begin{aligned} S &\Rightarrow A\alpha \mid \beta \\ A &\Rightarrow Sd \end{aligned}$$

after applying the above algorithm, should become

$$\begin{aligned} S &\Rightarrow A\alpha \mid \beta \\ A &\Rightarrow A\alpha d \mid \beta d \end{aligned}$$

and then, remove immediate left recursion using the first technique.

$$\begin{aligned} A &\Rightarrow \beta d A' \\ A' &\Rightarrow \alpha d A' \mid \epsilon \end{aligned}$$

Now none of the production has either direct or indirect left recursion.

Left Factoring

If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

$$A \Rightarrow \alpha\beta \mid \alpha\gamma \mid \dots$$

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.

Example

The above productions can be written as

$$\begin{aligned} A &\Rightarrow \alpha A' \\ A' &\Rightarrow \beta \mid \gamma \mid \dots \end{aligned}$$

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

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing $T[A, t] = \alpha$ with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

$$\alpha \rightarrow t \beta$$

That is α derives t (terminal) in the very first position. So, $t \in \text{FIRST}(\alpha)$.

Algorithm for calculating First set

Look at the definition of $\text{FIRST}(\alpha)$ set:

- if α is a terminal, then $\text{FIRST}(\alpha) = \{ \alpha \}$.
- if α is a non-terminal and $\alpha \rightarrow \epsilon$ is a production, then $\text{FIRST}(\alpha) = \{ \epsilon \}$.
- if α is a non-terminal and $\alpha \rightarrow \gamma_1 \gamma_2 \gamma_3 \dots \gamma_n$ and any $\text{FIRST}(\gamma_i)$ contains t then t is in $\text{FIRST}(\alpha)$.

First set can be seen as:

$$\text{FIRST}(\alpha) = \{ t \mid \alpha \xrightarrow{*} t \beta \} \cup \{ \epsilon \mid \alpha \xrightarrow{*} \epsilon \}$$

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Algorithm for calculating Follow set:

- if α is a start symbol, then $\text{FOLLOW}(\alpha) = \$$
- if α is a non-terminal and has a production $\alpha \rightarrow AB$, then $\text{FIRST}(B)$ is in $\text{FOLLOW}(\alpha)$ except ϵ .
- if α is a non-terminal and has a production $\alpha \rightarrow AB$, where $B \in \epsilon$, then $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(\alpha)$.

Follow set can be seen as: $\text{FOLLOW}(\alpha) = \{ t \mid S \xrightarrow{*} \alpha t^* \}$

Limitations of Syntax Analyzers

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.