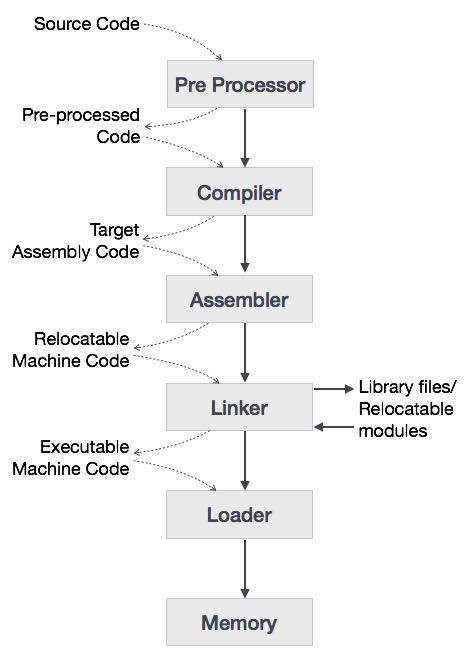
**Compiler Overview & Phases**

**Compiler Overview**

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

## Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

* User writes a program in C language (high-level language).
* The C compiler, compiles the program and translates it to assembly program (low-level language).
* An assembler then translates the assembly program into machine code (object).
* A linker tool is used to link all the parts of the program together for execution (executable machine code).
* A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

### Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

### Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

### Assembler

An assembler translates assembly language programs into machine code.The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

### Linker

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

### Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

### Cross-compiler

A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.

### Source-to-source Compiler

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

**Phases of Compiler Construction**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



### Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

< Token\_Name,Token\_Class>

### Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

### Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

### Symbol Table

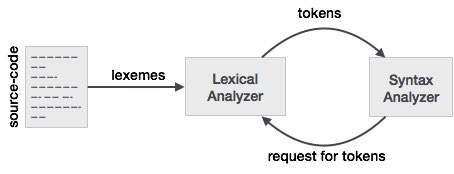
It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

# Lexical Analysis

**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 |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 Expression**

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 U M = {s | s is in L or s is in M}

* Concatenation of two languages L and M is written as

LM = {st | s is in L and t 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) U 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 { e, x, xx, xxx, xxxx, … }

* x+ means one or more occurrence of x.

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

* x? means at most one occurrence of x

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

[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 (qf)
* Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ 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(q0, qf), Σ(0,1), q0, qf, δ}

**Context Free Grammar**

**Context Free Grammar :- Definition**

***Definition*** − A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where

* **N** is a set of non-terminal symbols.
* **T** is a set of terminals where **N ∩ T = NULL.**
* **P** is a set of rules, **P: N → (N ∪ T)\***, i.e., the left-hand side of the production rule **P** does have any right context or left context.
* **S** is the start symbol.

**Classification of Context Free Grammars**

**C**ontext **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:

1) Based on number of strings it generates.

* If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)
* If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous grammar must not be used for parsing.  
   
2) Based on number of derivation trees.

* If there is only 1 derivation tree then the CFG is unambiguous.
* If there are more than 1 derivation tree, then the CFG is [ambiguous](https://www.geeksforgeeks.org/ambiguous-grammar/).

**Examples of Recursive and Non-Recursive Grammars**

**Recursive Grammars**

* 1. **S 🡪 S a S**

**S 🡪 b**

* 1. **S🡪 DV ;**

**D -> int | float | char**

**V 🡪 V,id | id**

**Non-Recursive Grammars**

1. **S 🡪 A a**

**A 🡪 b| c**

1. **S** 🡪 **DV**

**D 🡪 int | float | char**

**V 🡪 id**

**Types of Recursive Grammars**

Based on the nature of the recursion in a recursive grammar, a recursive CFG can be again divided into the following:

* Left Recursive Grammar (having left Recursion)
* Right Recursive Grammar (having right Recursion)
* General Recursive Grammar(having general Recursion)

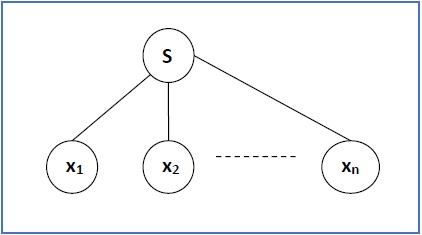
## Generation of Derivation Tree

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

### Representation Technique

* **Root vertex** − Must be labeled by the start symbol.
* **Vertex** − Labeled by a non-terminal symbol.
* **Leaves** − Labeled by a terminal symbol or ε.

If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows −



There are two different approaches to draw a derivation tree −

**Top-down Approach −**

* Starts with the starting symbol **S**
* Goes down to tree leaves using productions

**Bottom-up Approach −**

* Starts from tree leaves
* Proceeds upward to the root which is the starting symbol **S**

### Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

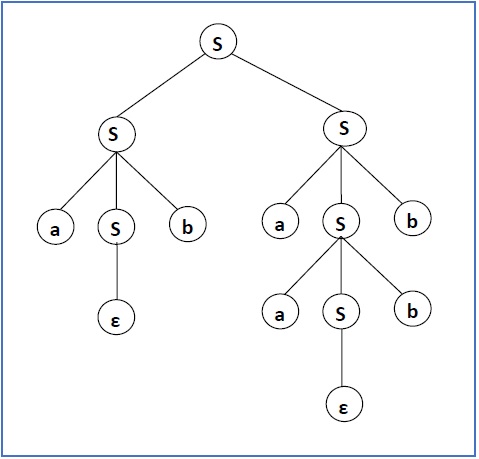
**Example**

Let a CFG {N,T,P,S} be

N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε

One derivation from the above CFG is “abaabb”

S → SS → aSbS → abS → abaSb → abaaSbb → abaabb



### Leftmost and Rightmost Derivation of a String

* **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
* **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

**Syntax Analysis**

**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 | w = wR } is not a regular language. But it can be described by means of CFG, as illustrated below:

G = ( V, Σ, P, S )

Where:

V = { Q, Z, N }

Σ = { 0, 1 }

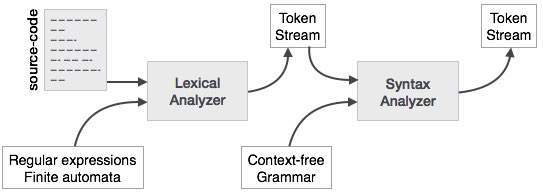
P = { Q → Z | Q → N | Q → ℇ | Z → 0Q0 | N → 1Q1 }

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 → E \* E

E → E + E \* E

E → id + E \* E

E → id + id \* E

E → id + id \* id

Step 1:

|  |  |
| --- | --- |
| E → E \* E | Parse Tree Construction |

Step 2:

|  |  |
| --- | --- |
| E → E + E \* E | Parse Tree Construction |

Step 3:

|  |  |
| --- | --- |
| E → id + E \* E | Parse Tree Construction |

Step 4:

|  |  |
| --- | --- |
| E → id + id \* E | Parse Tree Construction |

Step 5:

|  |  |
| --- | --- |
| E → id + id \* id | Parse Tree Construction |

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.

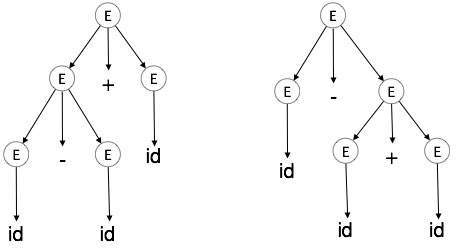
**Example**

E → E + E

E → E – E

E → id

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

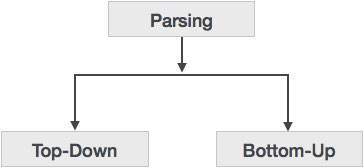


The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and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

**Parsing**

**Types of Parsing**

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



### Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

* **Recursive descent parsing** : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
* **Backtracking** : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

### Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

**Example:**

Input string : a + b \* c

Production rules:

S → E

E → E + T

E → E \* T

E → T

T → id

Let us start bottom-up parsing

a + b \* c

Read the input and check if any production matches with the input:

a + b \* c

T + b \* c

E + b \* c

E + T \* c

E \* c

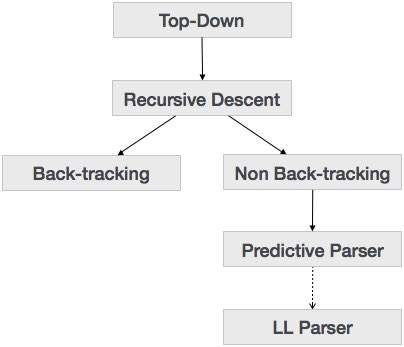
E \* T

E

S

**Top Down Parsing**

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



### Recursive Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

### Back-tracking

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd

X → oa | ea

Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

Now the parser matches all the input letters in an ordered manner. The string is accepted.

|  |  |  |  |
| --- | --- | --- | --- |
| Back Tracking | Back Tracking | Back Tracking | Back Tracking |