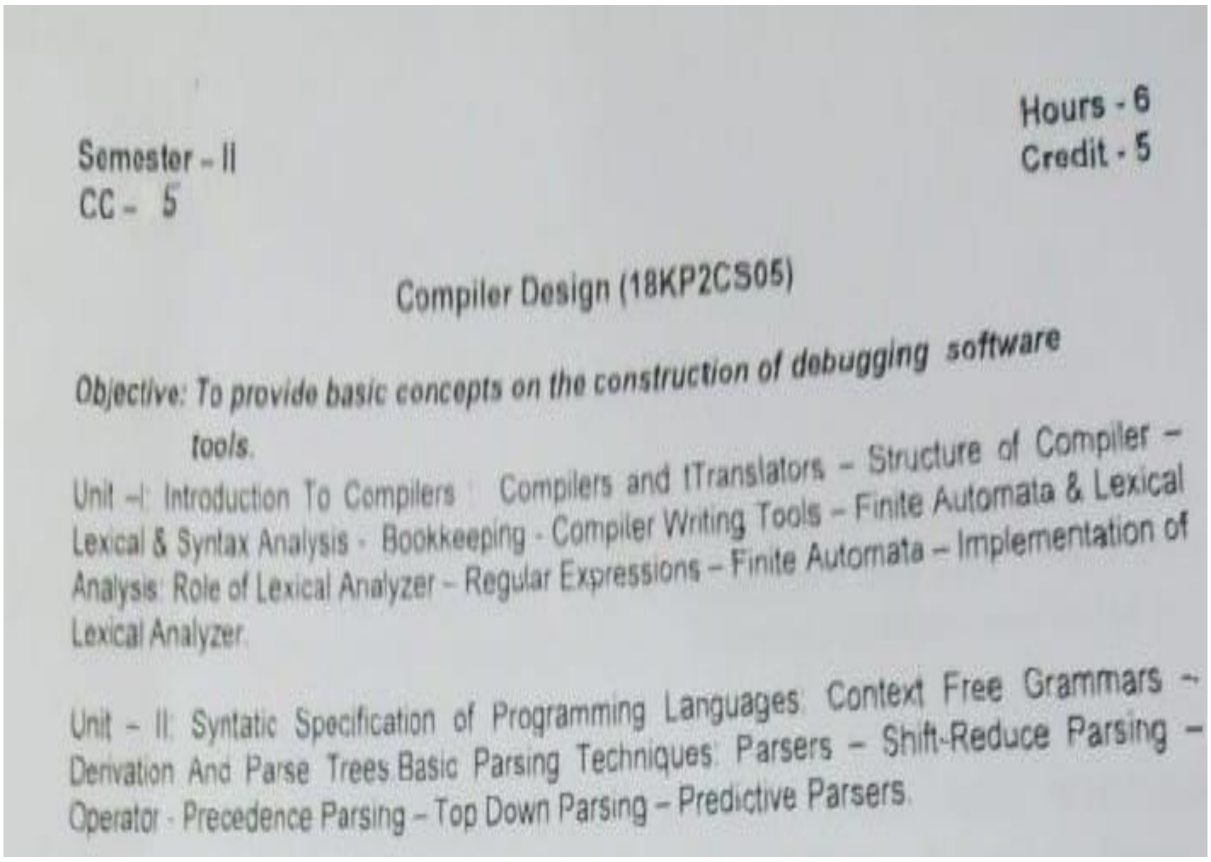
# DEPARTMENT OF COMPUTER SCIENCE

**SUBJECT NAME: COMPILER DESIGN**

**SUBJECT CODE: 18KP2CS05**



Semester – II
CC – 5

Hours - 6
Credit - 5

Compiler Design (18KP2CS05)

*Objective: To provide basic concepts on the construction of debugging software tools.*

Unit –I: Introduction To Compilers : Compilers and tTranslators – Structure of Compiler – Lexical & Syntax Analysis - Bookkeeping - Compiler Writing Tools – Finite Automata & Lexical Analysis: Role of Lexical Analyzer – Regular Expressions – Finite Automata – Implementation of Lexical Analyzer.

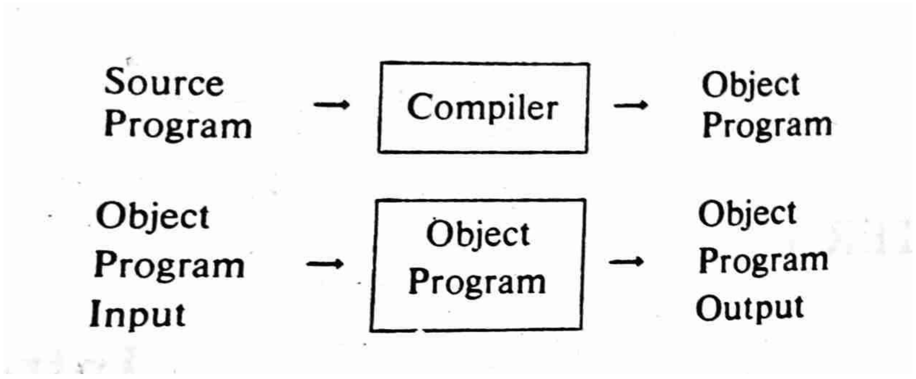
Unit – II: Syntatic Specification of Programming Languages: Context Free Grammars – Derivation And Parse Trees.Basic Parsing Techniques: Parsers – Shift-Reduce Parsing – Operator - Precedence Parsing – Top Down Parsing – Predictive Parsers.

**STAFF INCHARGE :**

**Mrs.K.HEMALATHA**

# INTRODUCTION TO COMPILERS

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Compilation and Execution

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

**ASSEMBLER**
Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language.
Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

**INTERPRETER**
An interpreter is a program that appears to execute a source program as if it were machine language.

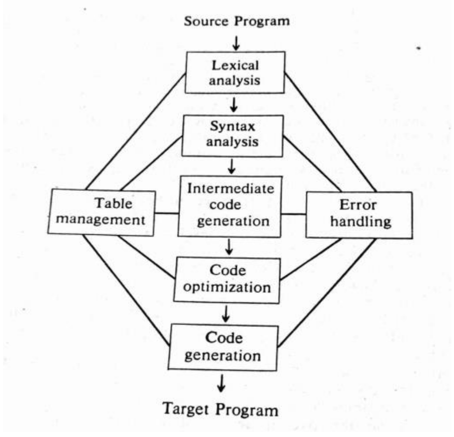
**TRANSLATOR**
A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:
1 Translating the HLL program input into an equivalent ml program.
2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

**LIST OF COMPILERS**

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

## STRUCTURE OF THE COMPILER

***Phases of a compiler:*** A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation.
The
phases of a compiler are shown in below
There are two phases of compilation.
a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)
Compilation process is partitioned into no-of-sub processes called **'phases'**.



**Lexical Analysis:-**
LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called token**s.**

**Syntax Analysis:-**
The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the

results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

**Intermediate Code Generations:-**
An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

**Code Optimization :-**
This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

**Code Generation:-**
The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

**Table Management (or) Book-keeping:-** This is the portion to **keep the names** used by the
program and records essential information about each. The data structure used to record this information called a 'Symbol Table'.

**Error Handlers:-**
It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as expression. Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

**The parser has two functions**. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

**Passes:**
Portions of one or more phases are combined into a module calles pass.

## Compiler-Writing Tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Parser Generator –**
   It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.

   **2. Scanner Generator –**
   It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.

**3.Facilities for Code-Generation** –

A high level language suitable for specifying the generation of intermediate, assembely or object code is provided by the compiler-compiler.

## Roles of the Lexical analyzer

Lexical analyzer performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

## 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 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

Nondetermini sticfiniteau tomata (NFA) haveno restrictions on the labels of their ed ges. A symbol can la bel several edges out of the same stat e, and E, t he emptystring ,isapossible label.

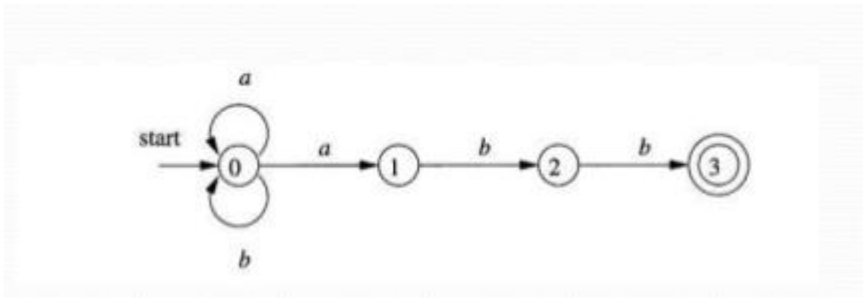
Deterministicfiniteautomata ( DFA) have, foreach stat e, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

**Nondeteministic FiniteAutomata**

- A finitesetof states S.
- A setof input symbolsC, the input a lpha bet .We assume that E, which stands for the empty string, is never a member of C.
- A t ransition function that g i v e s , for each state, and for each symbol in C U(E a set of next states.

A statesofrom S t hat isdist inguished asthe start state (or initial state).

A set of states F, a subset of S, that isdistinguished as t he accept ing states (or final states). The transi tiongraph for an NFA recognizing the language of regular expression (a/b)*abb is

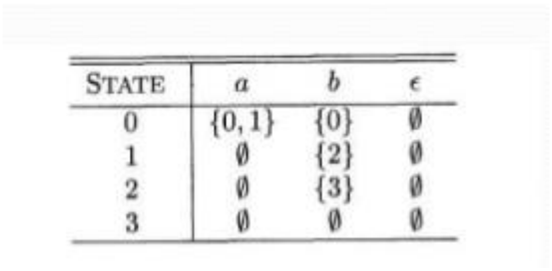


**Transistion Tables**

We can also represent an NF A by a transition table, whose rows correspond to states, and whose columns correspond to the input sym bols and c. The entry for a given state and input is the value of the transition funct ion applied to those arguments. If the transition function has no information about that state-input

pair, we put Q) in the table for the pair

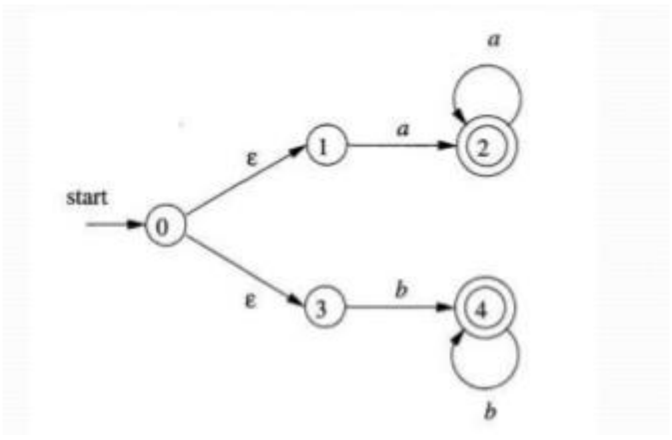
The Transistion Table ofNFA of Fig



| STATE | $a$ | $b$ | $\epsilon$ |
|---|---|---|---|
| 0 | {0,1} | {0} | ∅ |
| 1 | ∅ | {2} | ∅ |
| 2 | ∅ | {3} | ∅ |
| 3 | ∅ | ∅ | ∅ |

**Deterministic Finite Automata**

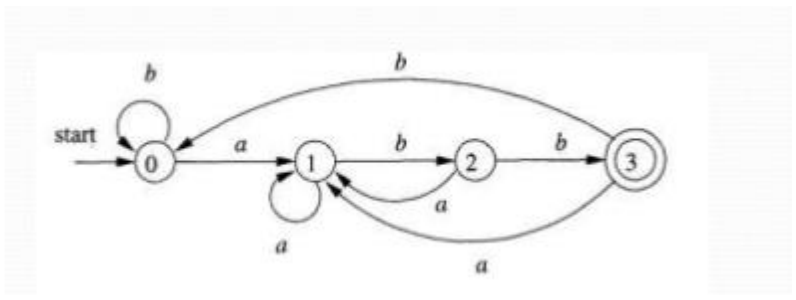
A deterministic finite automaton (DFA) is a special case of an N FA where :



**DFA**

- There are no moves on input E, and
- For each state sand input symbol a, there is exactly one edge out of s labeled a.

If we are using a transition table to represent a D FA, then each entry is a single state. we may therefore represent this state without the curly braces that we u se to form sets



**Implementation of Lexical analyser**

- Lexical analysis is the very first phase in the compiler designing
- A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token
- Lexical analyzer is implemented to scan the entire source code of the program
- Lexical analyzer helps to identify token into the symbol table
- A character sequence which is not possible to scan into any valid token is a lexical error
- Removes one character from the remaining input is useful Error recovery method
- Lexical Analyser scan the input program while parser perform syntax analysis
- It eases the process of lexical analysis and the syntax analysis by eliminating unwanted tokens
- Lexical analyzer is used by web browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS
- The biggest drawback of using Lexical analyzer is that it needs additional runtime overhead is required to generate the lexer tables and construct the tokens.

## SYNTAX SPECIFICATION OF PROGRAMMING LANGUAGES

The **syntax** of a **programming language** is usually described using a combination of the following two components: a regular expression describing its lexemes, and. a context-free grammar which describes how lexemes may be combined to form a syntactically correct **program**.

### Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

Where,

**G** describes the grammar

**T** describes a finite set of terminal symbols.

**V** describes a finite set of non-terminal symbols

**P** describes a set of production rules

**S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

**Example:**

L= {$wcw^R$ | w € (a, b)*}

**Production rules:**

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba
4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

## 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<br>E → E + id * id<br>E → id + id * id |

**Parse Tree**

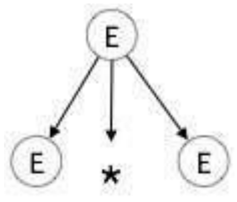
A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

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

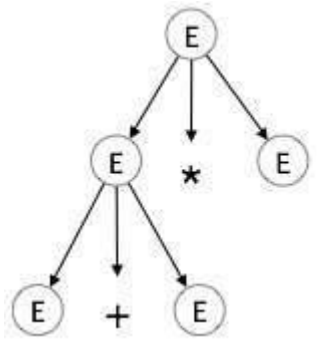
The left-most derivation is:

| |
|---|
| E → E * E<br>E → E + E * E<br>E → id + E * E<br>E → id + id * E |
| E → id + id * id |

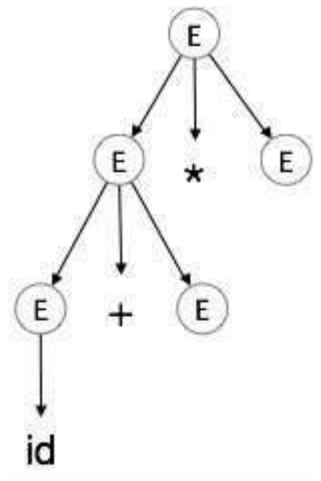
Step 1:

| | |
|---|---|
| E → E * E |  |

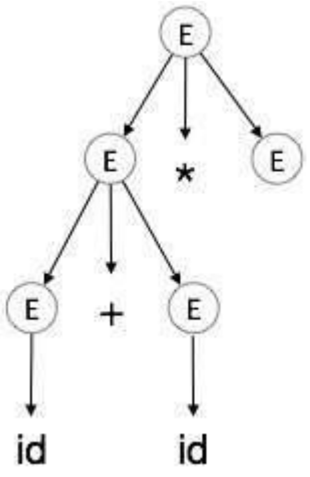
Step 2:

| | |
|---|---|
| E → E + E * E |  |

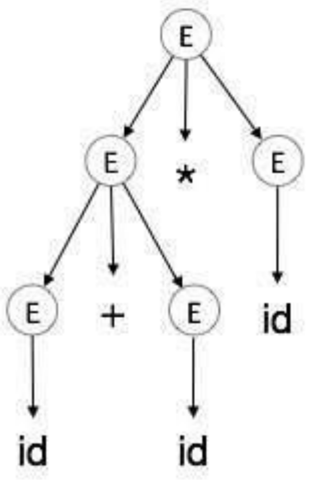
Step 3:

| | |
|---|---|
| E → id + E * E |  |

Step 4:

| | |
|---|---|
| E → id + id * E |  |

Step 5:

| | |
|---|---|
| E → 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.

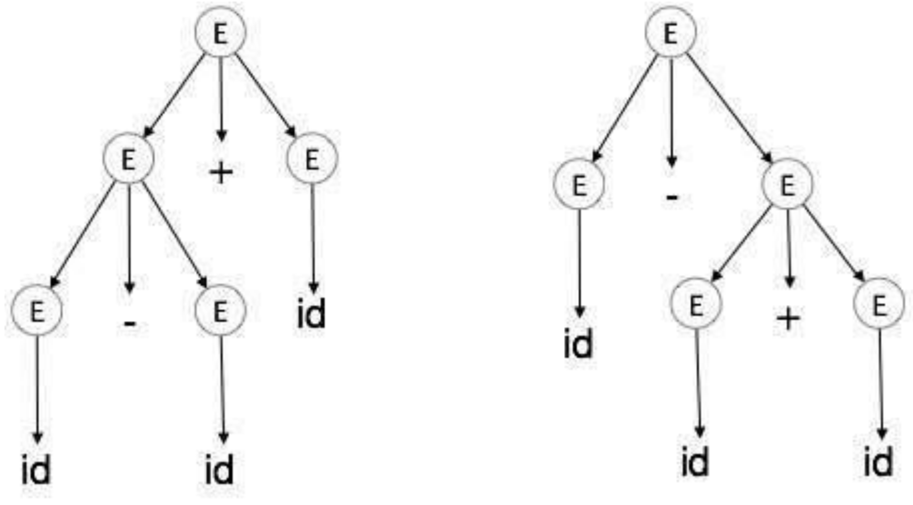
## Ambiguity

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 → 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.

## Shift Reduce Parser

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.

$$\text{A String} \xrightarrow[\text{reduce to}]{} \text{the starting symbol}$$

- Sift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduces parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

**Shift Reduce parser** attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.
This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

**Basic Operations –**

- **Shift:** This involves moving of symbols from input buffer onto the stack.
- **Reduce:** If the handle appears on top of the stack then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.
- **Accept:** If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.
- **Error:** This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

**Example 1 –** Consider the grammar

S –> S + S
S –> S * S
S –> id

Perform Shift Reduce parsing for input string "id + id + id".

| Stack | Input Buffer | Parsing Action |
|---|---|---|
| $ | id+id+id$ | Shift |
| $id | +id+id$ | Reduce S->id |
| $S | +id+id$ | Shift |
| $S+ | id+id$ | Shift |
| $S+id | +id$ | Reduce S->id |
| $S+S | +id$ | Reduce S->S+S |
| $S | +id$ | Shift |
| $S+ | id$ | Shift |
| $S+id | $ | Reduce S->id |
| $S+S | $ | Reduce S->S+S |
| $S | $ | Accept |

## Operator precedence parsing

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

Precedence table:

| | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|
| + | ⋗ | ⋖ | ⋖ | ⋗ | ⋖ | ⋗ |
| * | ⋗ | ⋗ | ⋖ | ⋗ | ⋖ | ⋗ |
| ( | ⋖ | ⋖ | ⋖ | ≐ | ⋖ | X |
| ) | ⋗ | ⋗ | X | ⋗ | X | ⋗ |
| id | ⋗ | ⋗ | X | ⋗ | X | ⋗ |
| $ | ⋖ | ⋖ | ⋖ | X | ⋖ | X |

Parsing Action

- Both end of the given input string, add the $ symbol.

- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example

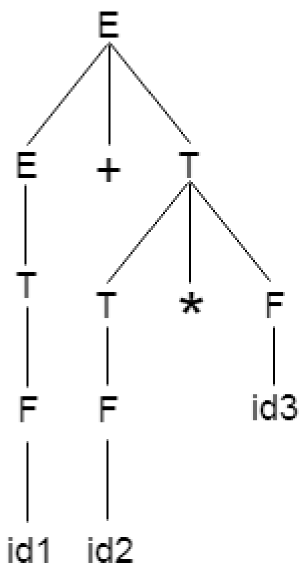
**Grammar:**

1. E → E+T/T
2. T → T*F/F
3. F → id

**Given string:**

1. w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

| | E | T | F | id | + | * | $ |
|---|---|---|---|---|---|---|---|
| E | X | X | X | X | ≐ | X | ⋗ |
| T | X | X | X | X | ⋗ | ≐ | ⋗ |
| F | X | X | X | X | ⋗ | ⋗ | ⋗ |
| id | X | X | X | X | ⋗ | ⋗ | ⋗ |
| + | X | ≐ | ⋖ | ⋖ | X | X | X |
| * | X | X | ≐ | ⋖ | X | X | X |
| $ | ⋖ | ⋖ | ⋖ | ⋖ | X | X | X |

Now let us process the string with the help of the above precedence table:

$\$ \lessdot \mathrm{id1} \gtrdot + \mathrm{id2} * \mathrm{id3}\ \$$

$\$ \lessdot F \gtrdot + \mathrm{id2} * \mathrm{id3}\ \$$

$\$ \lessdot T \gtrdot + \mathrm{id2} * \mathrm{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot \mathrm{id2} \gtrdot * \mathrm{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * \mathrm{id3}\ \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot \mathrm{id3} \gtrdot \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \doteq F \gtrdot \$$

$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

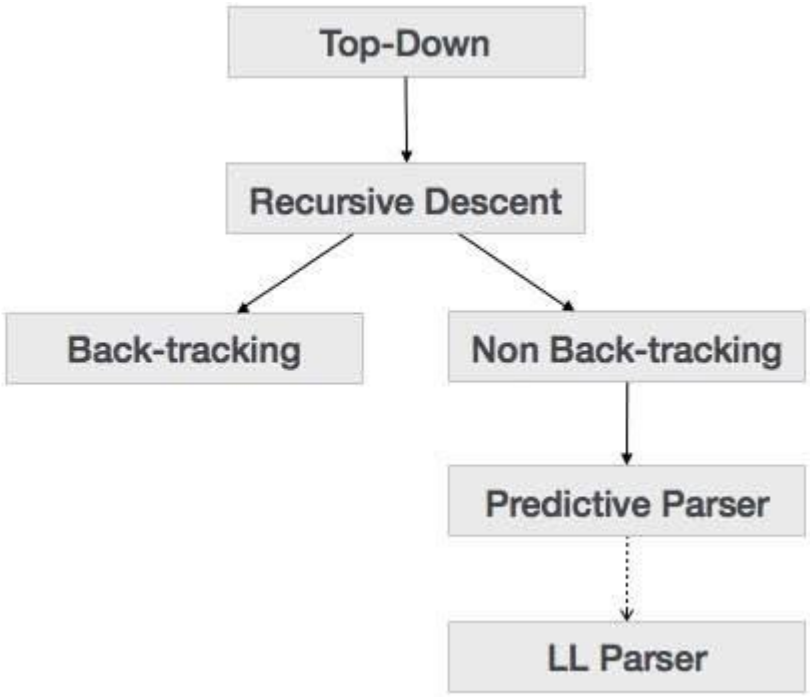
$\$ \lessdot E \doteq + \doteq T \gtrdot \$$

$\$ \lessdot E \gtrdot \$$

Accept.

## Top-Down Parser

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

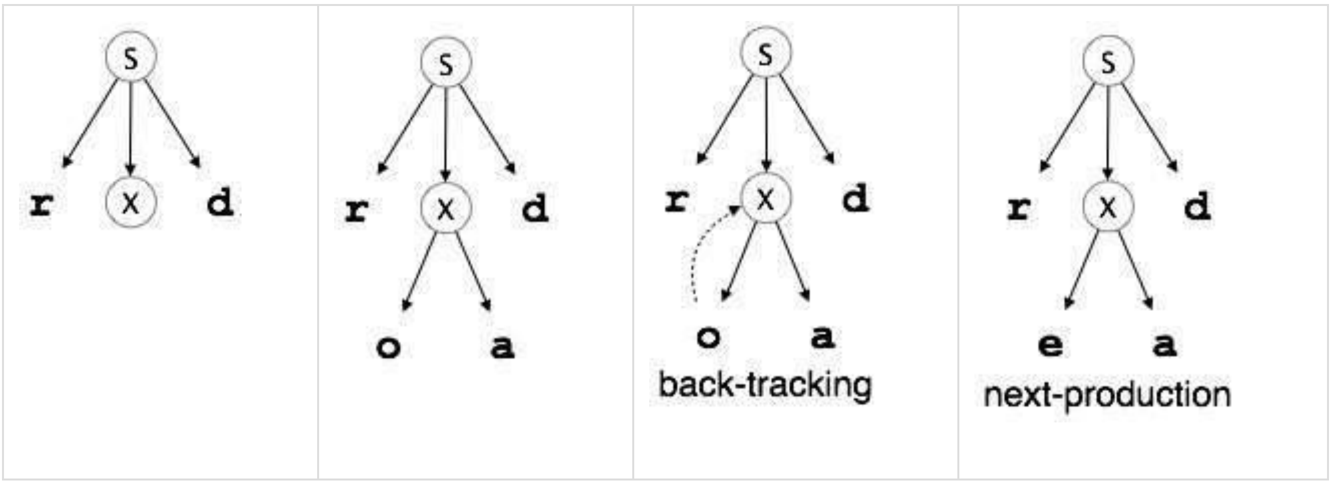
Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

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).

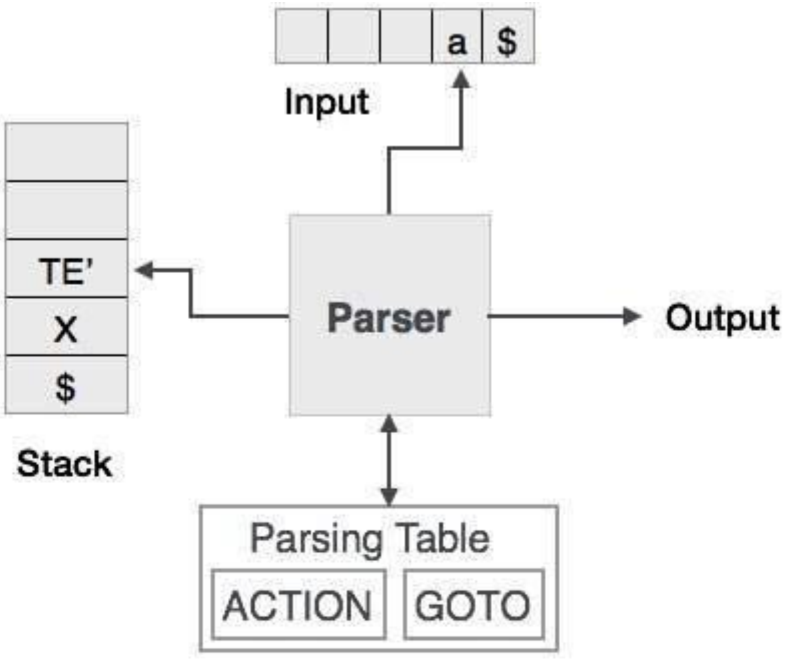
Now the parser matches all the input letters in an ordered manner. The string is accepted.



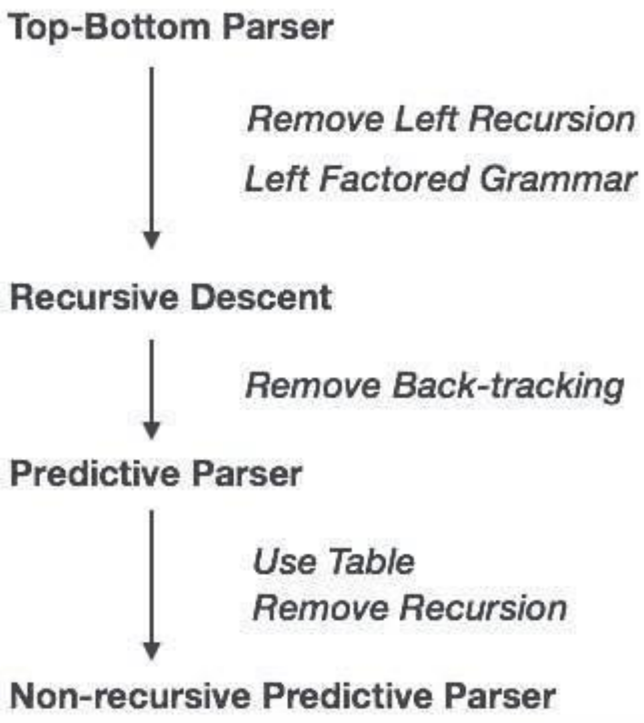
## Predictive Parser

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

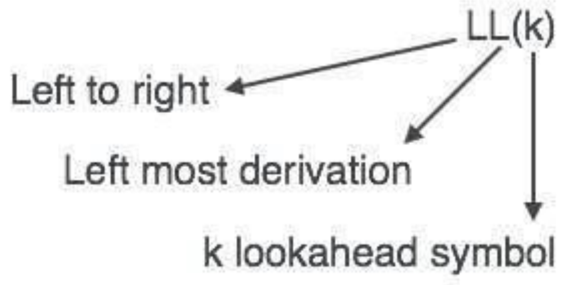


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

**LL Parser**

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



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.

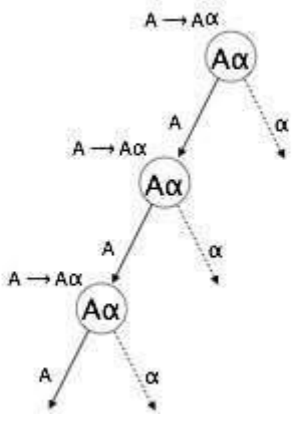
**Example:**

(1) A => Aα | β

(2) S => Aα | β
A => 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

$A \Rightarrow \beta A'$
$A' \Rightarrow \alpha A' \mid \varepsilon$

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 A1, A2, A3,…, $A_n$

  for each i from 1 to n
    {
    for each j from 1 to i-1
      {
      replace each production of form $A_i \Rightarrow Aj\gamma$
      with $A_i \Rightarrow \delta 1\gamma \mid \delta 2\gamma \mid \delta 3\gamma \mid \ldots \mid \gamma$
      where $A_j \Rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_n$ are current $A_j$ productions
      }
    }
  eliminate immediate left-recursion

END
```

**Example**

The production set

S => Aα | β
A => Sd

after applying the above algorithm, should become

S => Aα | β
A => Aαd | βd

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

A  => βdA'
A' => αdA' | ε

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 ⟹ αβ | α$\boldsymbol{\gamma}$ | …

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

A => αA'
A'=> β | $\boldsymbol{\gamma}$ | …

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] = α 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,

α → t β

That is α derives t (terminal) in the very first position. So, t ∈ FIRST(α).

*Algorithm for calculating First set*

Look at the definition of FIRST(α) set:

- if α is a terminal, then FIRST(α) = { α }.
- if α is a non-terminal and α → Ɛ is a production, then FIRST(α) = { Ɛ }.
- if α is a non-terminal and α → $\boldsymbol{\gamma}$1 $\boldsymbol{\gamma}$2 $\boldsymbol{\gamma}$3 … $\boldsymbol{\gamma}$n and any FIRST($\boldsymbol{\gamma}$) contains t then t is in FIRST(α).

First set can be seen as:

$$\mathrm{FIRST}(\alpha) = \{\, t \mid \alpha \xrightarrow{*} t\beta \,\} \cup \{\, \varepsilon \mid \alpha \xrightarrow{*} \varepsilon \}$$

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 FOLLOW() = $
- if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S *αt*}