LAB 1

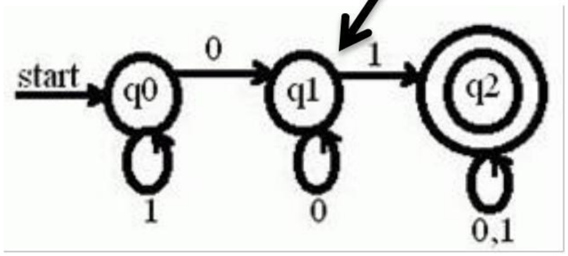
AIM: Implement a Deterministic Finite Automata (DFA) for a language.

Objective: To understand how a string is being processed from one state to other.

Theory:
A finite automaton is a 5-tuple M = ($\boldsymbol{Q}$, $\sum$, δ, $\boldsymbol{q_0}$,$\boldsymbol{F}$), where
Q is the finite set of states, $\sum$ is the finite alphabet set, δ is the Transition Function, $\boldsymbol{q_0} \subset Q$ is the initial state and $F \subset Q$ is the set of final states.

L (M) = the language of machine M= set of all strings machine M accepts.

Transition Diagram and Transition Table



Q	0	1
qo	**q1**	**q0**
q1	**q1**	**q2**
q2	**q2**	**q2**

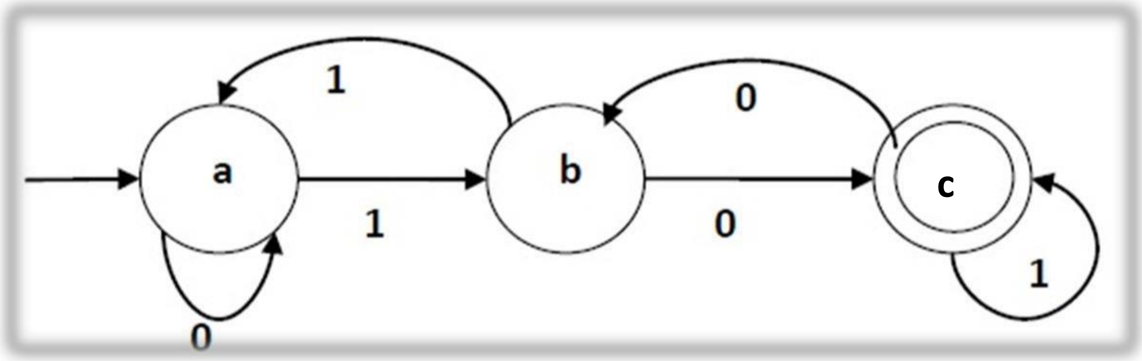
Finite Automaton can be classified into two types −

- Deterministic Finite Automaton (DFA)
- Non-deterministic Finite Automaton (NDFA / NFA)

Deterministic Finite Automaton (DFA)

In DFA, for each input symbol, one can determine the state to which the machine will move. Hence, it is called Deterministic Automaton. As it has a finite number of states, the machine is called Deterministic Finite Machine or Deterministic Finite Automaton.

Transition diagram



Here , Q = {a, b, c},
∑ = {0, 1},
q_0 = {a},
F = {c}, and

Transition Table

∑	**0**	**1**
a	a	b
b	c	a
c	b	c

Algorithm:

Step1: Create a 2D table T.

Step1.1: Declare state as one dimension.

Step1.2: Declare Input symbol as another one dimension.

Step2: For every transition $\mathbf{S_i} \xrightarrow{a} \mathbf{S_k}$ define
T[i,a] = k.

Spep3: Declare one more one dimensional array for input string named as ***input***.

Step4: Specify the initial state and final states.

Step5: Scan every input symbol from the ***input*** and check the state from the transition table T.

Pseudo Code:

```
//input index – i
//current state – state
…
…
.                 // Create the transition and input tables
.                 // Ask the states and string information from the user
.
i = 0;
state = 0;
while(input[i]){
      state = T(state,input[i++]);
}
…
.
.                 // Check the existence of state in the Final State array
….
```

LAB ASSIGNMENT 1 DATE:

Title: Implement a Deterministic Finite Automata (DFA) which accepts set of all strings over ∑= {*a, b*} of length ≤*n*, where *n*≥0.

a) **Provide the transition table.**
b) **Check whether a given string accepted or rejected.**
c) **Show the sequence of transitions.**

LAB 2

AIM: To implement lexical analyzer.

Objective: To show how a sequence of a string is represented as a token.

Theory: Lexical analyzer is used to analysis lexemes which is nothing but a sequence of strings.

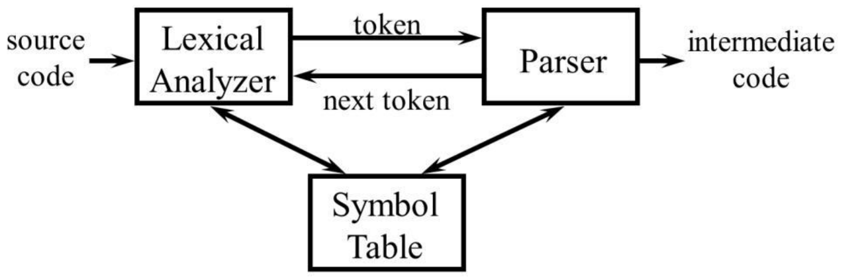


Fig 1: Lexical Analyzer

Token (language): a set of strings

- if, identifier, relop

Pattern (grammar): a rule defining a token

- if: if
- identifier: letter followed by letters and digits
- relop: < or <= or = or <> or >= or >

Lexeme (sentence): a string matched by the pattern of a token

- if, Pi, count, <, <=

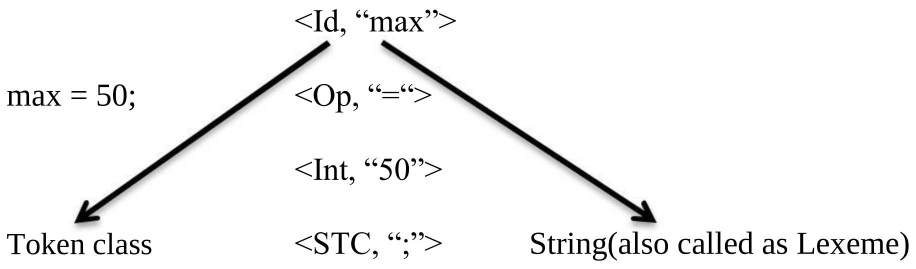
Attributes are used to distinguish different lexemes in a token

- < if, >
- < identifier, pointer to symbol table entry >
- < relop, ‘=’ >
- < number, value >

The role of the Lexical analyzer is to:-

Classify program substrings according to the token class.

Communicate tokens to the parser.



Token Class: Operator, Whitespace, Keyword, Identifier, Number

Special Token Class(STC): ( ); = (Also called as single character string)

An Implementation must do the following two things:-

1. Recognize **substrings** corresponding to tokens: **The lexemes**

2. Identify the **token class** of each lexeme: Create a separate array for all token class

Step1: Partitioning the string: Read the string left to right and recognize one token at a time.

Step1.1: Perform ***Lookahead*** to decide where one token ends and the next token begins.

Step2: Identify the **Token class :** Write the separate function to compare the token from the array of token class

LAB ASSIGNMENT 2 DATE:

Title: Consider any C program and identify the tokens by separating the followings into:

i) Identifier	**ii) Keyword**
ii)Operators	**iv)Special Symbol**

Also calculate how many numbers of tokens are present in the program.

LAB 3

AIM: To implement Recursive Descent Parsing

Objective: To understand the meaning of grammar by parsing from top to bottom.

Theory:

- Recursive descent parsing is a top down parsing technique.
- Consists of a set of procedures one for each non-terminal.
- Execution begins with the procedures, one for each non_terminal.

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings

Formally, a CFG is defined as G= (*V*, *T*, *P*, *S*)

Where,

V= a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.

T= a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.

P=a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).

S= a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminal have been replaced by terminal symbols.

Generally, CFG are grammar (V,T,S,P) having the production rule **P: V → (V ∪ T)***

Example: The grammar with production: P: A → aA, A → abc.

Here V={A}, T={a,b,c}, S={A}

Implementation of Recursive Descent Parsing

Input: Grammar G and String S.

Output: Successful or Error Message

S→ cAd   A →ab|a String: "cad"

Method:

S(){

```
    if input[i] = 'c' then {
        i++;
        if A() then{
            if input[i] = 'd' then{
                i++;
                return true;
            }
        }
    }
    return false;
}
```

Left Recursion:

A grammar is left recursive if it has a non-terminal ***A*** such that there is a derivation ***A=> Aα.***

Top down parsing methods can’t handle left-recursive grammars

A simple rule for direct left recursion elimination:

For a rule like: A→Aα|β
We may replace it with A→βα’ , A’→αA’| ε

The following grammar which generates arithmetic expressions

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathrm{id}$$

has two left recursive productions. Applying the above trick leads to

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid \mathrm{id}$$

Left recursion elimination algorithm:

```
–Arrange the nonterminal in some order 1, 2,..., .
–For (each i from 1 to n)
 {
     For (each j from 1 to i-1)
     {
          Replace each production of the form  $A \to A\alpha|\beta$ by the production $A \to \beta\alpha'$
          $A' \to \alpha A'|\ \varepsilon$
 }
Eliminate left recursion among the given productions
}
```

LAB ASSGNMENT 3 DATE:

Title: From a Context Free Grammar (CFG), write a program to:

- **i) Generate the variables and terminals present in the grammar**
- **ii) Eliminate Left recursion from the grammar**

LAB 4

AIM: To implement Top down parsing technique-Recursive Predictive techniques

Objective: To create a parse tree from the root towards the leaves scanning input from left to right.

Theory:

- Recursive Predictive Parsing is like Recursive Descent parsing.
- The Parser can predict which production to use by:-
    - Looking at the next few tokens – lookahead.
    - Using Restricted Grammar (Remove Left Recursion and Left Factoring – Manually).
    - No backtracking (pre deterministic parser).

Procedure for making a Predictive Parser:

1. Make a transition diagram (like DFA/NFA) for every rule of the grammar.
2. Optimize the DFA by reducing the number of states, yielding the final transition diagram.
3. To parse a string, simulate the string on the transition diagram.
4. If after consuming the input the transition diagram reaches an accept state, it is parsed.

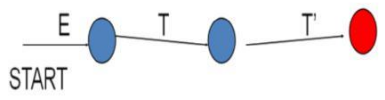
<u>To make Transition diagram:</u>

After removing left-recursion:

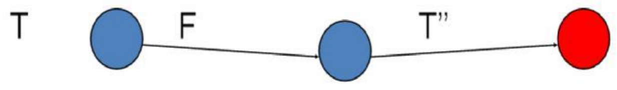
E→T T’
T’→+T T’ | ε
T→F T’’
T’’→*F T’’ |**ε**
F→E) |id

Step 1: make DFA-like transition diagrams for each rule:

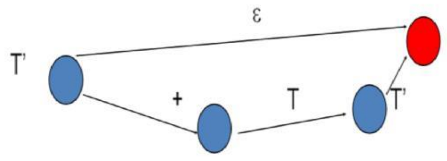
a) E→T T’



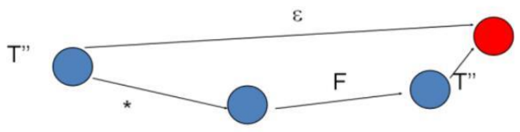
b) T→F T’’



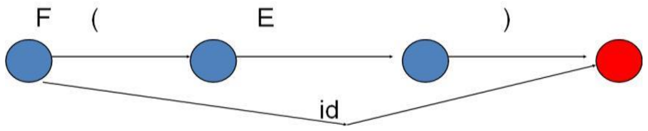
c) T’→+T T’ | ε



d) T’’→*F T’’ |**ε**



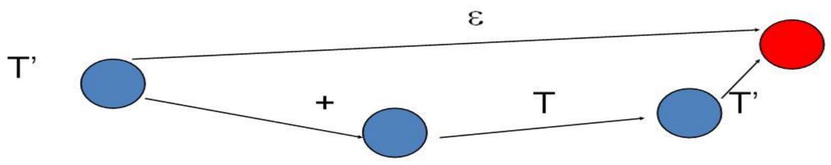
e) F → (E) |id



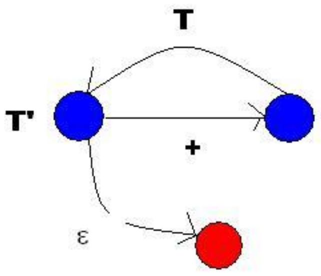
Step 2 : Optimization

The purpose of optimization is to reduce the number of states

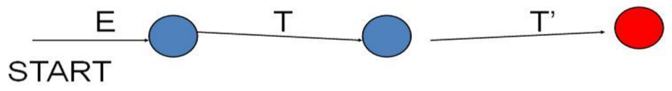
Let’s optimize the states. Consider the DFAfor **T’ →+T T’ | ε** shown below:



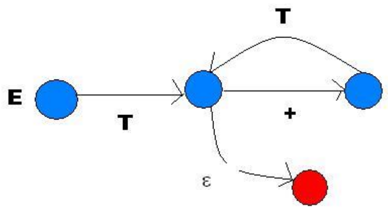
It can be optimized to



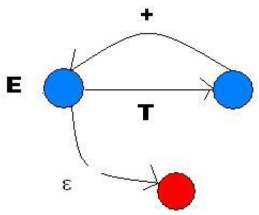
Which can be optimized further by combining with DFA for **E→T T’** shown below



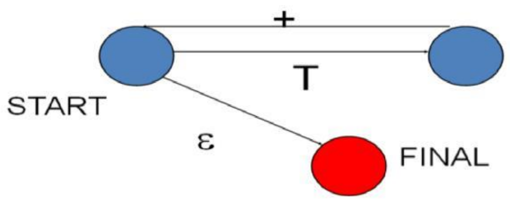
It yields

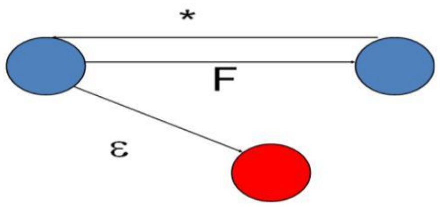


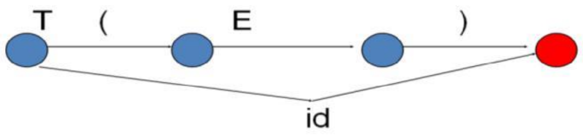
We can even further optimize it to produce



Similarly, we optimize other structures to produce the following DFA’s:







After optimization is over we start simulation over these “DFA’s”.

Step 3: Simulation on the input string:

Following steps list out the simulation procedure briefly.

Start from the start state

If a terminal comes consume it, move to next state

If a non – terminal comes go to the state of the “DFA” of the non-term and return on reaching the final state

Return to the original “DFA” and continue parsing

If on completion (reading input string completely), you reach a final state, and string is successfully parsed.

Algorithm:

Input: Grammar ***G*** and String S**:**

Output: Successful or Error Message

Step1: Remove the left recursion and left factoring from the given grammar G.

Step2: For each non-terminal do the following (Do Manually).

Step2.1: Create initial and final states of Transition Diagram.

Step2.2: For each production A →x1x2………xn create a path from initial state to final state with edges labeled x1,x2,………xn.

Step3: Write the procedures for all non-terminals obtained in step2..

Consider the grammar: E →E+T|T
T →T*F|F
F → (E)|id

```
E(){
    Call T();
    if input[i] = '+' then{
            i++;
            return true;
    }//-------Need to call T() recursively. See the Transition Diagram.
}
T(){
    call F();
    if input[i] = '*' then{
            i++;
            return true;
    } //-------Need to call F() recursively. See the Transition Diagram.
}
F(){
    if input[i] = '(' then{
            i++;
            call E();
            if input[i] = ')' then {
                    i++;
                    return true;
            }
    }
    else{
            if input[i] = 'id' then{
                    i++;
                    return true;
            }
    }
    return false;
}

main(){
    Read the string and attach $ at its end.
    if E() then
            print "success";
    else print "error";
}
```

LAB ASSIGNMENT 4 DATE:

Title: Consider a CFG without left recursion to implement Recursive Predictive Parsing Technique.

bexpr→bexpr or bterm | bterm
bterm→bterm and bfactor | bfactor
bfactor→not bfactor | (bexpr) | true | false

a) Remove left recursion

b) Draw the transition diagram

c) Check whether a given string “*not(true or false)*” is accepted or rejected.

LAB 5

AIM: To generate First set of a CFG.

Objective: To find the first symbol of a grammar.

Theory:

First () is set of terminals that begin strings derived from:

1. If $\alpha \Rightarrow \varepsilon$ then is also in First(ε)
2. In predictive parsing when we have **A** $\rightarrow$ **α|β**, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input

Algorithm: Calculating FIRST sets.

Step1: If X is a terminal, FIRST(X) is {X}.
Step2: If X is a non-terminal and $X \rightarrow \varepsilon$ is a production then add ε to FIRST (X).
Step 3: If X is a non-terminal and $X \rightarrow Y_1Y_2\ldots..Y_k$ is a production then add 'a' to FIRST(X). If for some 'i', 'a' is in FIRST (Yi) and ε is in all of FIRST (Y_1), FIRST (Y_2)…..FIRST (Y_{i-1}).
Step 4: If ε is in FIRST (Y_j), for all j, add ε to FIRST(X).

An Example

E →T E'
E' →+ T E' | ε
T →F T'
T' →* F T' | ε
F → (E) | id

FIRST(F) = { (, id }

FIRST(T') = { *, *ε* },

FIRST(T) = { (, id }

FIRST(E') = { +, *ε* },

FIRST(E) = { (, id }

LAB ASSIGNMENT 5

DATE:

Title: Write a program to generate the first set of a given grammar.

$$\begin{aligned}
S &\rightarrow aBDh \\
B &\rightarrow cC \\
C &\rightarrow bC \mid \varepsilon \\
D &\rightarrow EF \\
E &\rightarrow g \mid \varepsilon \\
F &\rightarrow f \mid \varepsilon
\end{aligned}$$

LAB 6

AIM: To generate the Follow set from a grammar (CFG).

Objective: To find the symbol(terminals or non-terminal) that appear after it.

Theory:

Define ***FOLLOW***(***A***), for nonterminal ***A***, to be the set of terminals ***α*** that can appear immediately to the right of ***A*** in some sentential form, that is, the set of terminals ***α*** such that there exists a derivation of the form ***S⇒ αAβ*** for some ***α*** and ***β*** . Note that there may, at some time during the derivation, have been symbols between ***A*** and ***α***, but if so, they derived ***ε*** and disappeared. If ***A*** can be the rightmost symbol in some sentential form, then ***$***, representing the input right endmarker, is in ***FOLLOW(A).***

Algorithm:

To compute ***FOLLOW(A)*** for all nonterminals ***A***, apply the following rules until nothing can be added to any ***FOLLOW*** set.

Step 1: Place ***$*** in ***FOLLOW(S)***, where ***S*** is the start symbol and ***$*** is the input right endmarker.

Step 2: If there is a production ***A ⇒ αBβ***, then everything in ***FIRST(β)***, except for **ε** is placed in ***FOLLOW(B).***

Step 3: If there is a production ***A ⇒ αβ***, or a production ***A ⇒ αBβ*** where ***FIRST (β)*** contains **ε** (i.e., ***β ⇒ ε***), then everything in ***FOLLOW (A)*** is in ***FOLLOW (B)***

Example:

Consider the expression grammar:

E →T E’
E’→+ T E’ | ε
T → F T’
T’→* F T’ | ε
F → (E) | id

Then,
FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E’) = {+, ε}
FIRST(T’) = {*, ε}
FOLLOW(E) = FOLLOW(E’) = {) , $}
FOLLOW(T) = FOLLOW(T’) = {+,),$}
FOLLOW(F) = {+, *,), $}

LAB ASSIGNMENT 6 **DATE:**

Title: Write a program to generate the Follow set of a given grammar.

$$
\begin{array}{lll}
S & \rightarrow & aBDh \\
B & \rightarrow & cC \\
C & \rightarrow & bC \mid \varepsilon \\
D & \rightarrow & EF \\
E & \rightarrow & g \mid \varepsilon \\
F & \rightarrow & f \mid \varepsilon
\end{array}
$$

LAB 7

AIM: To construct predictive parsing table.

Objective: To construct a top-down parser that never backtracks

Theory:

The following algorithm collects the information from ***FIRST*** and ***FOLLOW*** sets into a predictive parsing table ***M [A,a]*** , a two-dimensional array, where ***A*** is a nonterminal, and ***a*** is a terminal or the symbol ***$***, the input endmarker. The algorithm is based on the following idea:

- The production ***A → α*** is chosen if the next input symbol ***α*** is in ***FIRST(α).***
- The only complication occurs when α = ε or, more generally, α ∗ ⇒ε. In this case, we should again choose A → α, if the current input symbol is in FOLLOW (A), or if the $ on the input has been reached and $ is in FOLLOW (A).

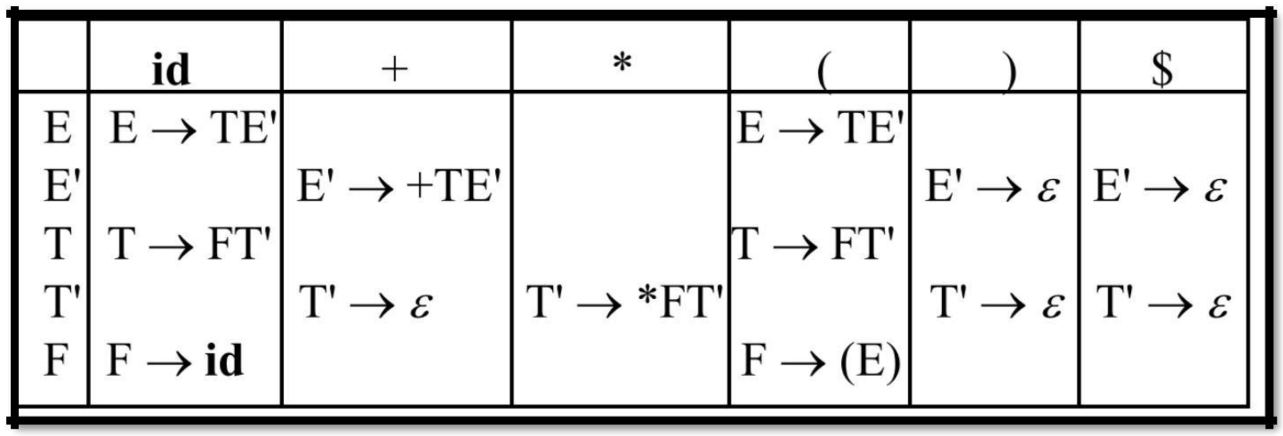
Algorithm:

Input: Grammar G.
Output: Parsing Table M.
Method.
Step 1: For each production ***A → α,*** do steps 2 and 3.
Step 2: For each terminal ***a*** in ***FIRST(α)***, add ***A → α*** to ***M[A, a].***
Step 3: If ***ε*** is in ***FIRST(α),*** add ***A → α to M[A, b]*** for each symbol ***b*** in ***FOLLOW(A).***
Step 4: If ***ε*** is in ***FIRST(α)*** and ***$*** is in ***Follow(A***), add ***A → α*** to ***M[A,$].***
Step 5: Make each undefined entry of ***M*** be error.

EXAMPLE:

Non Terminal	**FIRST ()**	**FOLLOW()**
E	{(,id}	**{), $}**
T	{(,id}	{+,), $}
E’	{+,ε}	{), $}
T’	{*,ε}	{+,), $}
F	{(,id}	{+, *,), $}

Predictive parsing table for the grammar is shown below:



	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Table 1: LL (1) Parsing Table

LAB ASSIGNMENT 7 **DATE:**

Title: Write a program to construct the predictive parsing table for a given grammar.

$$\begin{aligned}
S &\rightarrow aBDh \\
B &\rightarrow cC \\
C &\rightarrow bC \mid \varepsilon \\
D &\rightarrow EF \\
E &\rightarrow g \mid \varepsilon \\
F &\rightarrow f \mid \varepsilon
\end{aligned}$$

LAB 8

AIM: To implement non-recursive predictive parsing technique.

Objective: To efficiently implement by handling the stack of activation record explicitly.

Theory:

Predictive parsers are those recursive descent parsers needing no backtracking.
Grammars for which we can create predictive parsers are called ***LL(1)***

–The first ***L*** means scanning input from left to right
–The second ***L*** means leftmost derivation
–And ***1*** stands for using one input symbol for lookahead

A grammar ***G*** is ***LL(1)*** if and only if whenever ***A*** $\rightarrow$ ***α|β***are two distinct productions of ***G***, the following conditions hold:

- For no terminal a do ***α*** and ***β*** both derive strings beginning with ***a***
- At most one of ***α*** or ***β*** can derive empty string∴.
- If β ⇒ ε, then a does not derive any string beginning with a terminal in FOLLOW(A).
- Likewise, if α ⇒ε, then β does not derive any string beginning with a terminal in FOLLOW(A).

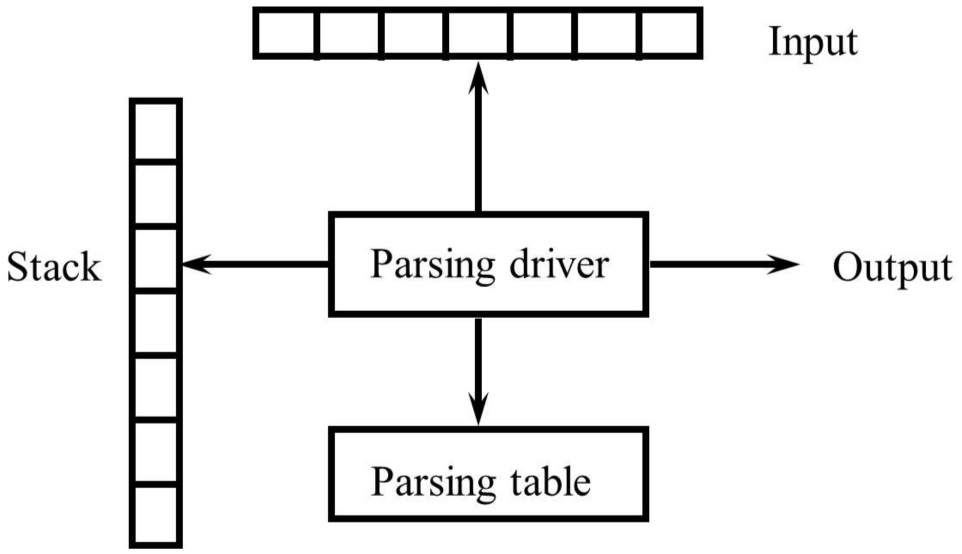


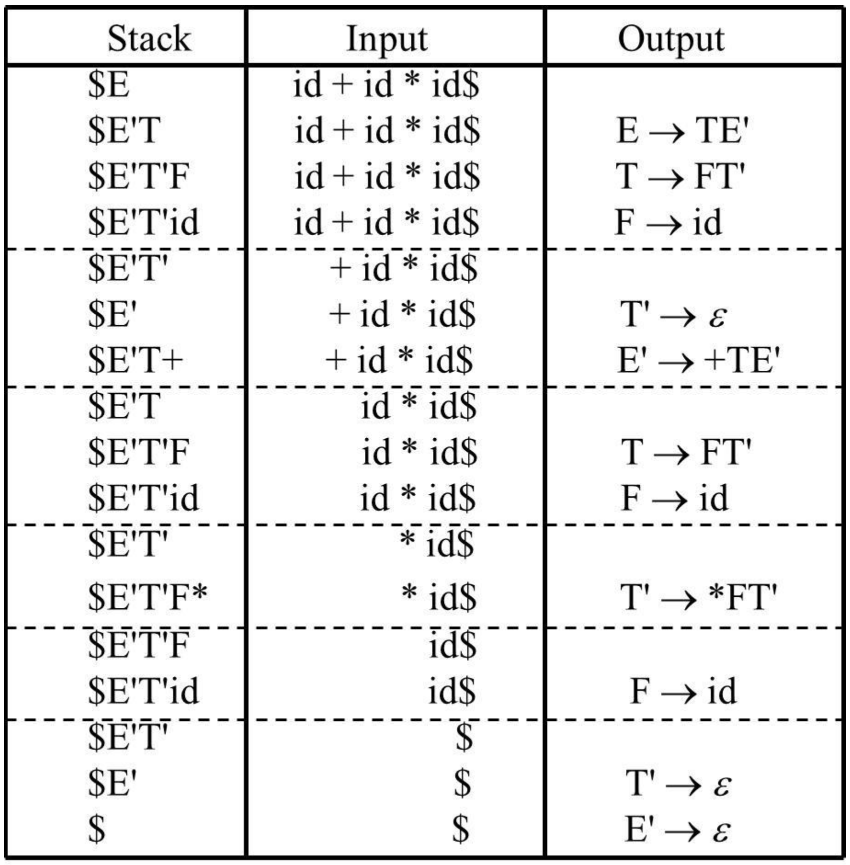
Fig 2: Syntax Analyzer

For every LL(1) grammar , each parsing-table entry uniquely identifies a production or signals an error as shown in table 1.

Algorithm:

Input: String w and Parsing Table M **Output:**
Successful parsing or Error message. push $\$S$
onto the stack, where S is the start symbol set *ip*
to point to the first symbol of $w\$$;

```
repeat
   let $X$ be the top stack symbol and $a$ the symbol pointed to by $ip$;
   if $X$ is a terminal or $ then
      if $X = a$ then
         pop $X$ from the stack and advance $ip$
      else error
   else /* X is a nonterminal */
      if $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$ then
         pop $X$ from the stack and push $Y_k \dots Y_2 Y_1$ onto the
      stack else error
until $X = \$$ and $a = \$$
```



Stack	Input	Output
$E	id + id * id$	
$E'T	id + id * id$	$E \rightarrow TE'$
$E'T'F	id + id * id$	$T \rightarrow FT'$
$E'T'id	id + id * id$	$F \rightarrow id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \rightarrow \varepsilon$
$E'T+	+ id * id$	$E' \rightarrow +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \rightarrow FT'$
$E'T'id	id * id$	$F \rightarrow id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \rightarrow *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \rightarrow id$
$E'T'	$	
$E'	$	$T' \rightarrow \varepsilon$
$	$	$E' \rightarrow \varepsilon$

Table 2 shows the moves made by a predictive parser on input “***id+id*id***”

LAB ASSIGNMENT 8 DATE:

Title: Write a program to construct Non-Recursive Predictive Parser for the given grammar.

a) Take parsing table as input
b) Show stack implementation
c) Check whether the string *110110* could be parsed or not.

LAB 9

AIM: To remove left factor from a grammar.

Objective: To predict which rule to use without backtracking

Theory:

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:

  Stmt -> **if** expr **then** stmt **else** stmt | **if** expr **then** stmt

- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:

If we have $A \to \alpha\beta 1 \mid \alpha\beta 2$ then we replace it with

$A \to \alpha A'$

$A' \to \beta 1 \mid \beta 2$

Algorithm

- For each non-terminal A, find the longest prefix α common to two or more of its alternatives.

If $\alpha \neq \varepsilon$, then replace all of A-productions $A \to \alpha\beta 1 \mid \alpha\beta 2 \mid \dots \mid \alpha\beta n \mid \gamma$ ***by:***

$A \to \alpha A' \mid \gamma$

$A' \to \beta 1 \mid \beta 2 \mid \dots \mid \beta n$

Example:

stmt →**if** expr **then** stmt **else** stmt | **if** expr **then** stmt

Can be left factored as:

stmt →**if** expr **then** stmt X

X→else stmt| ε

LAB ASSIGNMENT 9 DATE:

Title: Write a program to remove left factor from a given grammar:

$A \rightarrow aAB \,/\, aBc \,/\, aAc$

LAB 10

AIM: To implement bottom-up parsing technique.

Objective: To use a stack to hold grammar symbols and an input buffer to hold the string to be parsed and to find the handle which matches the RHS of production that reduced to the non-terminal.

Theory:

Bottom-up parsing is based on the reverse process to top-down. Instead of expanding successive nonterminal according to production rules, a current string or right sentential form is collapsed each time until the start non-terminal is reached to predict the legal next symbol. This can be regarded as a series of reductions. The approach is known as **shift-reduce parsing.**

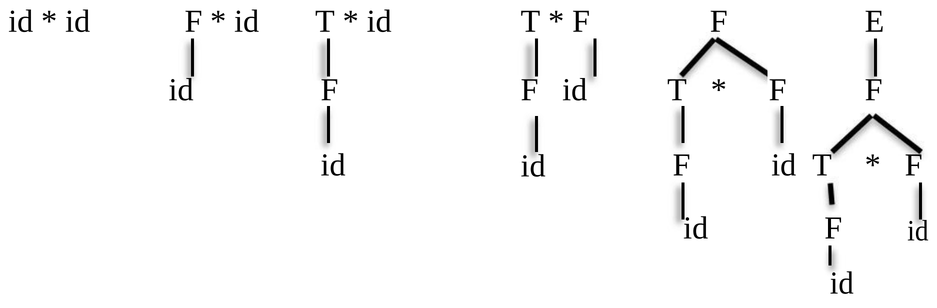
Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top).

Example: id*id

E →E + T | T

T →T * F | F

F →(E) | id



Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply

Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation.

- Suppose that the parser is currently working on right-sentential form F * id.
- The handle is F. That is the right-hand side of production $T \rightarrow F$.
  Replacing F by T yields T * **id**.
- Recursively can compare the right hand side of the production with the elements on top of the stack. This is how handles can be obtained.
- If successful then push non-terminal present on the left hand side of the production to the stack

Stack Operations

Shift: shift the next *input symbol* onto the top of the stack.
Reduce: replace the handle at the top of the stack with the corresponding *nonterminal*.
Accept: announce successful completion of the parsing.
Error: call an error recovery routine

Algorithm:
Input: An input string ***w***
Output: If ***w*** is well formed, a skeletal parse tree is generated otherwise an error indication.
Method: Initially stack contains ***$*** and input buffer contains string ***w$.***
Step1: Start
Step2: set **input** to point to the 1st symbol of ***w$***
Step3: repeat forever.

```
        Step 3.1: if $S is on the top of stack and input points to the $ then return.
        Step 3.2: else if
          Let $A1A2...........An is in the stack and a production exists B → A2........An i.e.
a handle exists then, reduce B in stack by
      $A1B. Step 3.3: else if
                There is no handle and ip points to a terminal
                 Shift the terminal on top of stack
       Step 3.4: else error();
       Step 3.5: end for
```

Step 4: stop.

LAB ASSIGNMENT 10 DATE:

Title: Write a program to implement the shift-reduce parsing for the given grammar, for the string “32423”.

E –> 2E2
E –> 3E3
E –> 4

a) Show the stack implementation

b) Check whether a string “23243” could be parsed or not.

c) Identify the handles.

LAB 11

AIM: To implement Operator Precedence Parsing Technique.

Objective: To find the handle of right-sentential form.

Theory:

Operator precedence parsing is a bottom-up parsing technique.

Define three disjoint precedence relation between certain pair of terminals.

a<b;	b has higher precedence than a
a=b;	b has same precedence as a
a>b;	b has lower precedence than a

	+	*	(	)	id	$
+	>	<	<	>	<	>
*****	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

Table: 3 Operator precedence table

Algorithm:

Input : An input string ***w$,*** stack initially containing ***$*** and a table containing precedence relations between certain terminal.

```
Step 1: start.
Step 2: set input to point to first symbol of w$.
Step 3: repeat forever.
    Step 3.1: if ($ on top of stack and input point to $ ) then return.
    Step 3.2: else
        {
```

```
                    Let ‘a’ be the topmost terminal symbol on the
stack and ‘b’ be the symbol pointed by input.
                    If (a< b or a=b) then                /*shift */
                    {
                        Push ‘b’ on to stack
                        Advance input to point to next symbol (i.e.
                        input symbol);
                     }

Else if (a > b) then                / *reduce/
                  Repeat pop stack
                        Until (top of stack terminal is related
                        by < to the terminal most recently
                        popped)
                 Else error
}
```

Step 4: Stop.

LAB ASSIGNMENT 11 DATE:

Title: Write a program to implement the operator precedence for the given grammar:

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L , S \mid S$$

a) Construct the operator precedence table.
b) Show the stack implementation.
c) Check whether the string "(a , (a , a))" is accepted or not.

LAB 12

AIM: To implement Intermediate code generation

Objective: To produce a code that would be able to translate into machine code.

Theory:
In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

If we generate machine code directly from source code then for n target machine we will have ***n*** optimizers and n code generators but if we will have a machine independent intermediate code, we will have only one optimizer. Intermediate code can be either language specific (e.g., Bytecode for Java) or language, independent (three-address code).

The following are commonly used intermediate code representation:

1. Postfix Notation –

- The ordinary (infix) way of writing the sum of ***a*** and ***b*** is with operator in the middle : ***a* + *b***
- The postfix notation for the same expression places the operator at the right end as ***ab*** +.
- In general, if ***e1*** and ***e2*** are any postfix expressions, and **+** is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by ***e1e2* +**. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example :
The postfix representation of the expression ***(a – b) * (c + d) + (a – b)*** is : ***ab – cd + ab -+****.

2. Three-Address Code –

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three

address code. Three address statement is of the form ***x = y op z*** , here ***x, y, z*** will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statements.
Example – The three address code for the expression ***a + b * c + d***:

T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d

T 1, T 2, T 3 are temporary variables.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms: ***quadruples, triples and indirect***.

r1 = c * d;
r2 = b + r1;
a = r2

Quadruples
Each instruction in quadruples presentation is divided into four fields*: operator, arg1, arg2, and result*. The above example is represented below in quadruples format:

Op	**arg1**	**arg2**	**result**
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples
Each instruction in triples presentation has three fields : *op, arg1, and arg2*.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

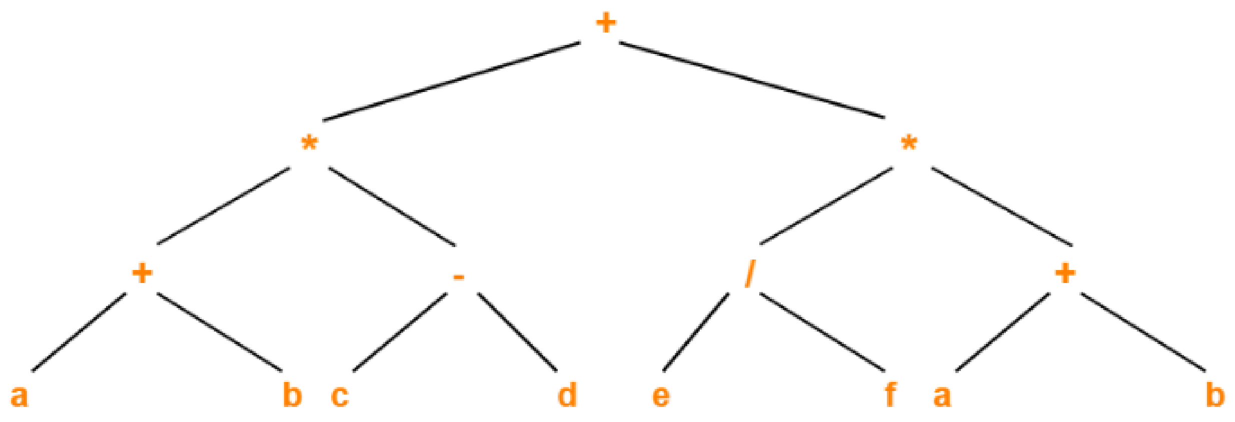
Op	arg1	arg2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results is positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code

3. Syntax Tree –



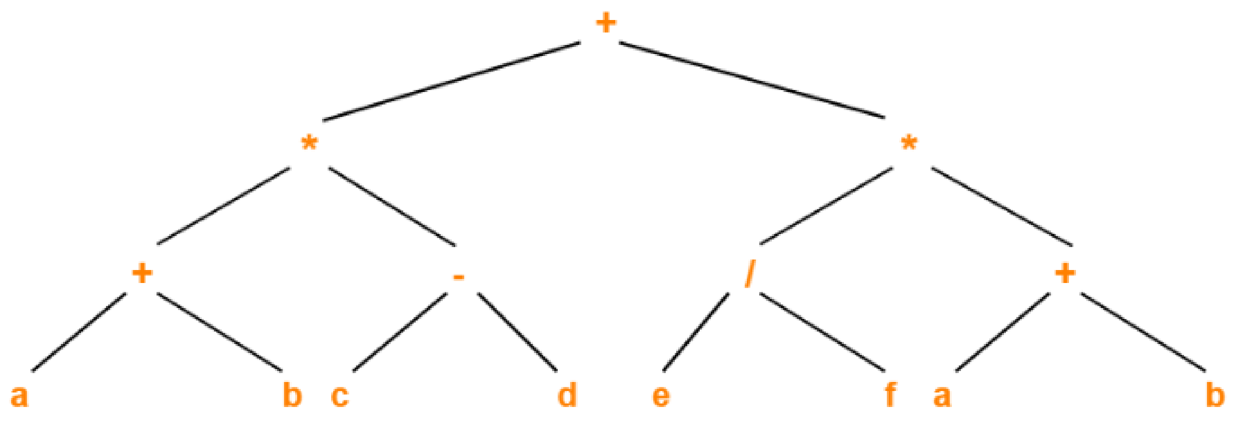


Fig 3: Syntax Tree

LAB ASSIGNMENT 12 **DATE:**

Title: For any given arithmetic expression: *a+b*c-d/(b*c)*

1. **Generate Three-Address-Code (TAC).**
2. **Implement TAC representation: Quadruple, Triple & Indirect Triples.**