COMPILER LAB VIVA QUESTIONS

1. **What is a compiler?**
   It is a tool or an application which converts high level language program to low level language program(Machine Language).

   e.g. Turbo C, gcc are compilers for C language.

2.**What is :Editor, Preprocessor,compiler, Assembler,Linker, Loader**

Editor

Editor is a tool used to write a program. e.g. gedit, Notepad, Wordpad, MS Word etc.

Preprocessor

Preprocessor expands the program. It adds preprocessor directives definitions in the program wherever they are called. e.g. header files, constant definitions, macros etc.

Compiler

It is important application. It converts high level language program to low level language program(Assembly Language/Machine Language). It includes six phases, that we will see in the next part of this post.

Assembler

It is a tool, which accepts assembly language code and converts it into the machine language(executable) code. This executable code has relocatable logical addresses.

Linker

It is the tool which links program to the global variables.

Loader

Loader loads the program from secondary storage(Hard disk, usb storage etc.) to the main memory(RAM) to get executed by the processor. Loader uses the relocatable logical addresses and save the executable code into the actual address space in the main memory.

3. **What is the difference between compiler and interpreter?**

Compiler	Interpreter
Compiler works on complete program at once i.e. Compiler take complete program as input.	Interpreter works line by line i.e. Interpreter takes one line as input at one time instance.
Compiler generates intermediate code i.e. object code.	Interpreter does not generate intermediate code.
Programs need to be compiled once and can be run any number of times.	Interpreter has to work every time for the same program.
Compiler takes more memory as it generates intermediate code which has to be saved in memory.	Interpreter is memory efficient as it does not generate intermediate object code.

4. **What are the six phases of compiler?**

LexicalAnalyzer

High level language program has several parts. e.g. keywords, macros, identifiers, operators etc. These are called as tokens. Lexical analyzer accepts the high level language program and tokenize it. These tokens are saved into the symbol table. To find out the tokens, lexical analyzer uses regular expressions(patterns). It matches the words from programs with regular expressions and identify whether which token it is.
For above example,

id=id1+id2*id3
(For simplicity, I have mentioned like this. We can also write it as id op id1 op1 id2 op2 id3 where id,id1,id2,id3 represents identifiers and op,op1,op2 represents operators).

SyntaxAnalyzer(Parser)

Syntax analyzer prepares a parse tree from input it got from above step. Parse tree is useful to decide the sequence of operations or priorities of operations.

Semantic Analyzer

Semantic analyzer checks the semantics of a expression(statement). This means, it checks whether all the entities in the statement are as per the rules or not. For example, we can not assign the new value to the constant or constant value should not be on the left hand side of a = operator. Such kinds of rules are checked by the semantic analyzer.

Intermediate Code generator

Intermediate code generator, generates code in terms of temporary variables.
For above example, intermediate code will be as follows,

```
temp1=id2*id3
temp2=id1+temp1
id=temp2
```

Code Optimizer

Code optimizer optimizes(in this case, reduces) the size of code if it is possible.
For above example,

```
temp1=id2*id3
id=id1+temp1
```

Code Generator

Code generator generates the assembly code in terms of registers from the input it got from above steps.
For above example (suppose id1,id2,id3 are saved to registers AX,BX,CX respectively), then assembly code will be

```
MUL BX,CX
ADD AX, CX
```

5. What is a cross compiler?

Cross compiler is a compiler which converts source program compatible for one architecture to the another program compatible for another architecture.

6. **What is front end and back end of a compiler?**

In compilers, the frontend translates a computer programming source code into an intermediate representation, and the backend works with the intermediate representation to produce code in a computer output language. The backend usually optimizes to produce code that runs faster.

7. **What is a symbol table and its use?**

Symbol Table is **an important data structure created and maintained by the compiler in order to keep track of semantics of variables** i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

8.**What is lexeme in compiler?**

A lexeme is a sequence of alphanumeric characters in a token. The term is used in both the study of language and in the lexical analysis of computer program compilation. In the context of computer programming, lexemes are part of the input stream from which tokens are identified.

9. **What is a lexeme?**

The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

10. **What is a token?**

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages. **Example of tokens:**

```
Keywords; Examples-for, while, if etc.
Identifier; Examples-Variable name, function name, etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc
```

11. **What are the three parts of a Lex Program**.

LEX program consists of three sections

1. "Definitions",
2. "Regular Expressions and action for each regular expression"
3. "Subroutines".

Definition section consists of C language code which involves header file inclusion, global variables declaration/definition etc. C language code can be mentioned in between the symbols **%{** and **%}**

LEX requires regular expressions or patterns to identify token of lexemes. Examples of token are identifier, header file, constants etc. while Lexemes are the actual words used in your input program for e.g. printf, scanf, stdio.h etc. These regular expressions and action for them are mentioned in second section. When we call **yylex()** function, it starts the process of pattern matching. Lex keep the matched string into the address pointed by pointer **yytext**.

Matched string's length is kept in **yyleng** while value of token is kept in variable **yylval.** Third section consists of subroutines/functions. Lex call **yywrap()** function after input is over. It should return 1 when work is done or should return 0 when more processing is required. **yylex()** function actual starts the process of pattern matching. **yyparse()** in yacc program automatically calls yylex(). Thats why there in no need to call yylex() separately. If you are writing standalone Lex program, then you have to call yylex() in main() function in Lex program.

12. **What are the three parts of a YACC Program?**

YACC(Yet Another Compiler Compiler) program consists of three sections: 1."Definitions", 2."Context Free Grammar and action for each production", 3."Subroutines/Functions".

In first section, we can mention C language code which may consist of header files inclusion, global variables/ Constants definition/declaration. C language code can be mentioned in between the symbols **%{** and **%}.** Also we can define tokens in the first section. We can define the **associativity** of the operations (i.e. left associativity or right associativity). **Priorities** among the operators can also be specified.

In second section, we mention the grammar productions and the action for each production.

Third section consists of the subroutines. We have to call **yyparse()** to initiate the parsing process. **yyerror()** function is called when all productions in the grammar in second section do not match to the input **statement.**

13.What is the use of yyparse()?

We have to call **yyparse()** to initiate the parsing process i.e. the process of checking syntax (i.e. process of matching grammer productions).

14.**What is yytext,yylval and yylength?**

Lex keep the matched string into the address pointed by pointer **yytext**. Matched string's length is kept in **yyleng** while value of token is kept in variable **yylval.**

15.**What is an ambiguous grammar?**

A CFG is said to be ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft**M**ost **D**erivation **T**ree (LMDT) or **R**ight**M**ost **D**erivation **T**ree (RMDT).

Let us consider this grammar: **E -> E+E|id** We can create a 2 parse tree from this grammar to obtain a string **id+id+id.** The following are the 2 parse trees generated by left-most derivation.

(e.g:) `E -> I`

```
E -> E + E
E -> E * E
```

```
E -> (E)
I -> ε | 0 | 1 | … | 9
```

From the above grammar String **3*2+5** can be derived in 2 ways:

```
I) First leftmost derivation                    II) Second leftmost
derivation
        E=>E*E                              E=>E+E
         =>I*E                               =>E*E+E
         =>3*E+E                                           =>I*E+E
         =>3*I+E                               =>3*E+E
         =>3*2+E                               =>3*I+E
         =>3*2+I                               =>3*2+I
         =>3*2+5                               =>3*2+5
```

16. Write intermediate code for a=b+c*d.
Intermediate code for a=b+c*d is as follows:

t1=c
t2=d
t3=t1*t2
t4=b
t5=t4+t3
a=t5

Optimized Intermediate code will be as follows:
t1=c
t2=d
t3=t1*t2
t4=b+t3
a=t4

17. **Explain Quadruple, Triple and Indirect Triple.**

Implementation of Three Address Code –
There are 3 representations of three address code namely
- Quadruple
- Triples
- Indirect Triples

<u>Quadruples-</u>
It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

In quadruples representation, each instruction is splitted into the following 4 different fields-

op, arg1, arg2, result

Here-

- The op field is used for storing the internal code of the operator.
- The arg1 and arg2 fields are used for storing the two operands used.
- The result field is used for storing the result of the expression.

Triples –

This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

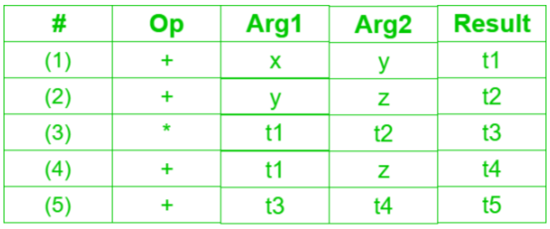
Indirect Triples –

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Question – Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

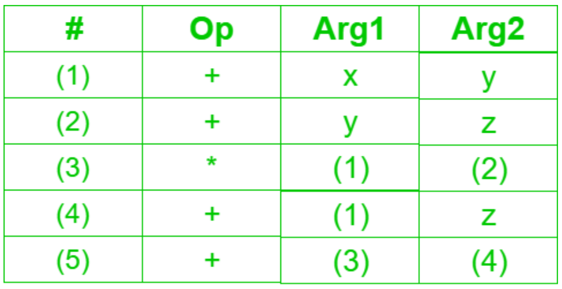
Explanation – The three address code is:

```
t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z

t5 = t3 + t4
```



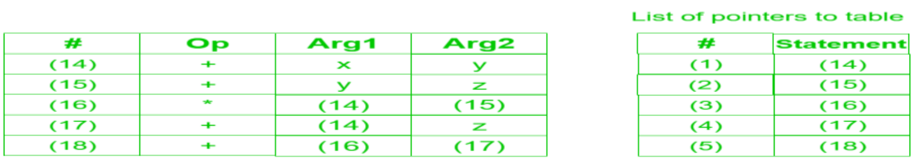
#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation



#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation



#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

List of pointers to table

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

18. **What is three address code?**

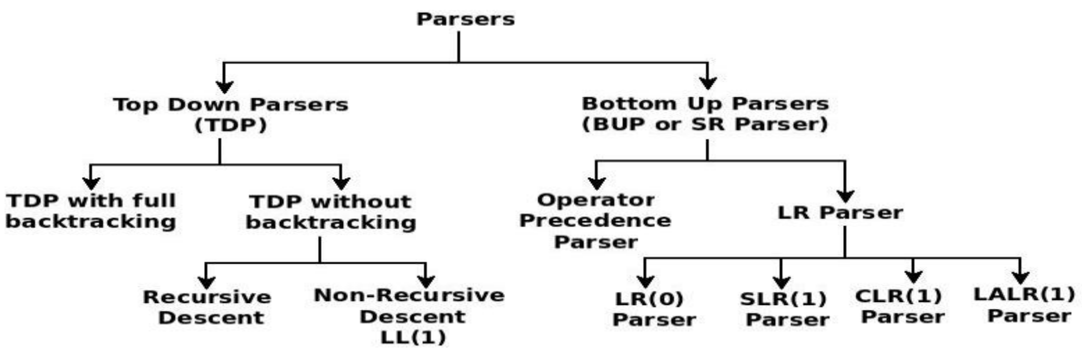
Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

General representation –

a = b op c

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator.

19. **What are the different types of parsers?**



Broadly, parsers can be classifies into two major categories:
1. Top Down Parser
2. Bottom Up Parser

In Top Down Parser, we start with a Starting Non-Terminal , we use the grammar productions and try to reach to the string.

Bottom Up Parsing is opposite to the Top Down Parsing. Here we start with String. We reduce it by using grammar productions and try to reach to the Start symbol.

I. Top Down Parser

Top Down Parsers can be categorized as follows:

i. TDP with full backtracking
ii. TDP without backtracking

TDP without backtracking is again categorized as follows:

a. Recursive Descent Parser
b. Non-recursive Descent Parser/ LL(1) Parser

II. Bottom Up Parser / SR Parser

Bottom Up Parsers can be categorozed as follows:

i. Operator Precedence Parser
ii. LR Parser

LR Parsers are again categorized as follows:

a. LR(0) Parser
b. SLR(1) Parser

c. LALR(1) Parser
d. CLR(1) Parser

[Note: Study the working of all the parsers]

20. **What is the Basic difference between LR(0) and SLR(1)?**

In LR(0), for a closure which has item of the form (A->BCD.) i.e. item which has Dot at the end, we have to add Reduce action <u>for all terminals</u>. That is why, it is also called to have zero Lookahead symbol.
While in SLR(1), for a closure which has item of the form (A->BCD.) i.e. item which has Dot at the end, we have to add Reduce action for <u>FOLLOW(A) terminals</u>. That is the only difference between LR(0) and SLR(1).
21. **What is the difference between top-down and bottom up parsing?**

In Top Down Parser, we start with a <u>Starting Non-Terminal</u> , we use the grammar productions and try to reach to the string.

Bottom Up Parsing is opposite to the Top Down Parsing. Here we start with String. We reduce it by using grammar productions and try to reach to the Start symbol.
22. What is difference between LR(0) item and LR(1) item?

In LR(0) and SLR(1), items which we use, are called LR(0) items which are of the form (A->.BCD or A->B.CD etc.). In CLR(1) and LALR(1), items which we use, are called as LR(1) items. These are of the form (A->.BCD, α). Here α is a Lookahead symbol which represents the terminals after A .

<u>How To find Lookahead Symbol?</u>
1. We start with item (S'->.S). It has Lookahead symbol $ as we assume that string ends at symbol $. So complete LR(1) item becomes (S'->.S, $).
2. If (A->.BCD, α) is item in closure, then next item will be (B->.DEF, FIRST(CDα).
Similarly,
If (A->B.CD, α) is item in closure, then next item will be (C->.GHI, FIRST(Dα).
If (A->BC.D, α) is item in closure, then next item will be (C->.JKL, FIRST(α).
3. When we have to find first item of next closure, then Lookahead symbols are retained from the previous closure.
23. How to find FIRST and FOLLOW of a grammar with examples

FIRST of a grammar:
A Non-terminal can generate a sequence of terminals(non-empty string) or empty string. The collection of initial terminal of all these strings is called a FIRST of a Non-terminal of a Grammar.
<u>How to find set FIRST(X):</u>
For all productions whose LHS is X,
1. If RHS starts with terminal, then add that terminal to the set FIRST(X).
2. If RHS is ϵ, then add ϵ to the set FIRST(X).
3. If RHS starts with Non-Terminal(say Y), then add FIRST(Y) to the set FIRST(X). If FIRST(Y) includes ϵ, then, also add FIRST(RHS except Y) to the set FIRST(X).

<u>What is FOLLOW of a Non-Terminal of a Grammar:</u>

In a derivation process, the collection of initial terminal(i.e. FIRST) of a string which follows Non-terminal, is called FOLLOW of that Non-terminal.

For finding FOLLOW set of a Non-Terminal, check in RHS of all productions which consist of that Non-Terminal.

<u>How to find set FOLLOW(X):</u>

1. If X is "Start" symbol, then add $ to the set FOLLOW(X). (Reason: Each string generated from grammar is assumed that it ends in $. For e.g. abc$ or a+b$. As that string can be generated from the Start symbol, Start symbol is followed by $).
2. If in any RHS, X is followed by terminal (say t), then add t to the set FOLLOW(X).
3. If in any RHS, X is followed by Non-terminal (say Y), then add FIRST(Y) except ϵ to the set FOLLOW(X). If FIRST(Y) contains ϵ, then you have to also add FIRST of <u>remaining part of RHS after Y</u> to the set FOLLOW(X). If <u>remaining part of RHS after Y</u> is empty, then add FOLLOW(LHS) to the set FOLLOW(X).
4. If X is the last symbol in any RHS (For e.g. Z->wX), then add FOLLOW(LHS) i.e. FOLLOW(Z) to the set FOLLOW(X). (Reason: RHS is derived from LHS. So whatever follows RHS, also follows LHS. As X is last symbol in RHS, FOLLOW(X) includes FOLLOW(LHS)).

<u>Examples:</u>

<u>1. Grammar:</u>

S->xyz/aBC
B->c/cd
C->eg/df

FIRST(S)={x,a}
FIRST(B)={c}
FIRST(C)={e,d}

FOLLOW(S)={$}
FOLLOW(B)={e,d}
FOLLOW(C)={$}

24. **What is 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.

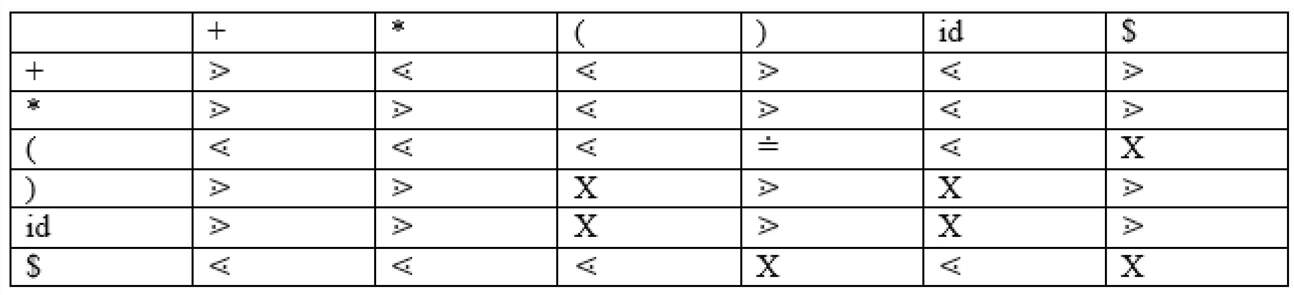
There are the three operator precedence relations:

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

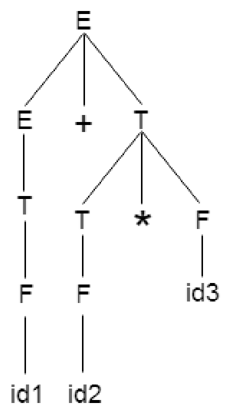
Grammar:

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	$\doteq$	X	$\gtrdot$
T	X	X	X	X	$\gtrdot$	$\doteq$	$\gtrdot$
F	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
id	X	X	X	X	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	X	$\doteq$	$\lessdot$	$\lessdot$	X	X	X
*	X	X	$\doteq$	$\lessdot$	X	X	X
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	X	X	X

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

$\$ \lessdot id1 \gtrdot + id2 * id3 \$$

$\$ \lessdot F \gtrdot + id2 * id3 \$$

$\$ \lessdot T \gtrdot + id2 * id3 \$$

$\$ \lessdot E \doteq + \lessdot id2 \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot F \gtrdot * id3 \$$

$\$ \lessdot E \doteq + \lessdot T \doteq * \lessdot id3 \gtrdot \$$

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

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

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

$\$ \lessdot E \gtrdot \$$

Accept.

25.**Explain recursive descent parsing?**

Recursive Descent Parser uses the technique of Top-Down Parsing without backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a Recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

The major approach of recursive-descent parsing is to relate each non-terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non-terminal, and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is prescribed by the productions for the equivalent non-terminal.

The recursive procedures can be simply to write and adequately effective if written in a language that executes the procedure call effectively. There is a procedure for each non-terminal in the grammar. It can consider a global variable lookahead, holding the current input token and a procedure match (Expected Token) is the action of recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. Match () is effectively a call to the lexical analyzer to get the next token.

For example, input stream is a + b$.

lookahead == a

match()

lookahead == +

match ()

lookahead == b

Example − Write down the algorithm using Recursive procedures to implement the following Grammar.

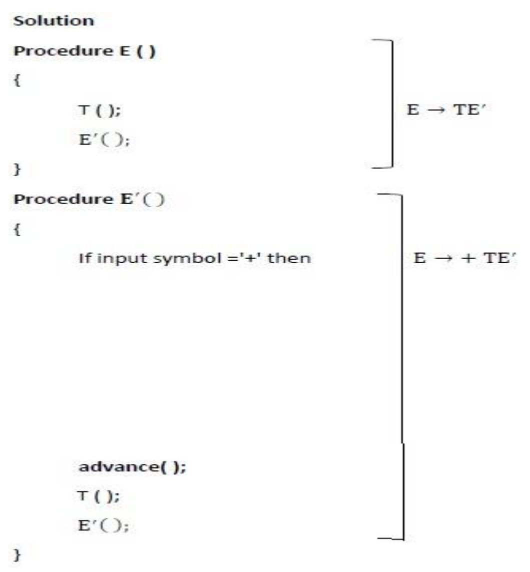
$E \rightarrow TE'$

$E' \rightarrow +TE'$

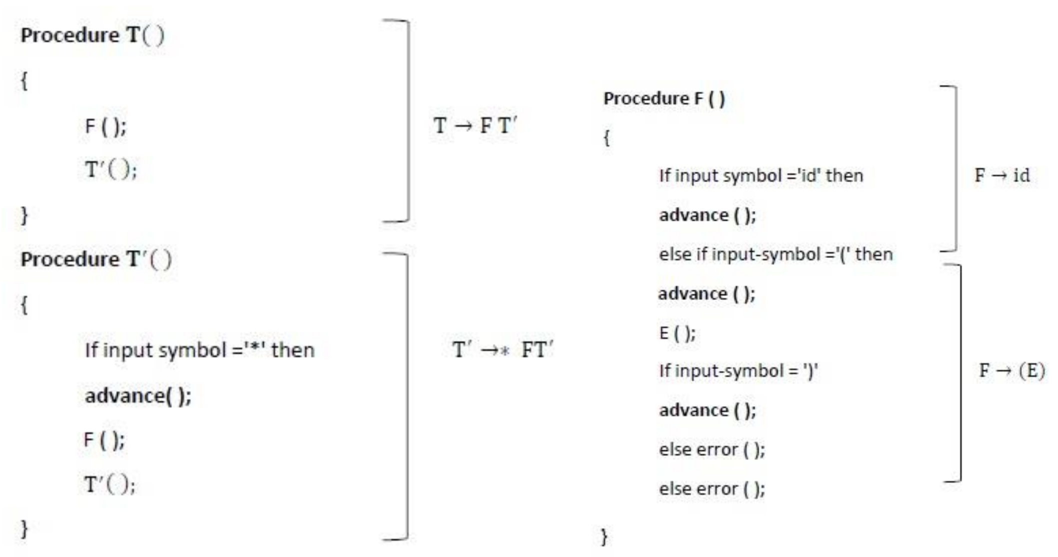
$T \rightarrow FT'$

$T' \rightarrow * FT'|\varepsilon$

$F \rightarrow (E)|id$



```
Solution
Procedure E ( )
{                                      E → TE′
        T ( );
        E′( );
}
Procedure E′( )
{
        If input symbol ='+' then      E → + TE′




        advance( );
        T ( );
        E′( );
}
```



```
Procedure T( )
{
        F ( );
        T′( );
}
```

$T \rightarrow F T'$

```
Procedure T′( )
{
        If input symbol ='*' then
        advance( );
        F ( );
        T′( );
}
```

$T' \rightarrow * \ FT'$

```
Procedure F ( )
{
        If input symbol ='id' then
        advance ( );
        else if input-symbol ='(' then
        advance ( );
        E ( );
        If input-symbol = ')'
        advance ( );
        else error ( );
        else error ( );
}
```

$F \rightarrow id$

$F \rightarrow (E)$

One of major drawback or recursive-descent parsing is that it can be implemented only for those languages which support recursive procedure calls and it suffers from the problem of left-recursion.

26. **Explain the steps in Shift reduce parsing?**

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 the shift-reduce parser is the LR parser.
This parser requires some data structures i.e.

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

The basic operations are:

- **Shift:** This involves moving symbols from the 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 a production rule is popped out of a stack and LHS of a production rule is pushed onto the stack.
- **Accept:** If only the start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accepted 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

27. what is left recursion? How is it eliminated?

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

$A \rightarrow A\ \alpha\ |\beta.$

The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$A \rightarrow \beta A'$

$A \rightarrow \alpha A'|\epsilon$

Example1 − Consider the Left Recursion from the Grammar.

$E \rightarrow E + T|T$

$T \rightarrow T * F|F$

$F \rightarrow (E)|id$

Eliminate immediate left recursion from the Grammar.

Solution

Comparing $E \rightarrow E + T|T$ with $A \rightarrow A\ \alpha\ |\beta$

E	→	E	+T	|	T
A	→	A	α	|	Β

$\therefore A = E, \alpha = +T, \beta = T$

$\therefore A \rightarrow A\ \alpha\ |\beta$ is changed to $A \rightarrow \beta A'$and $A' \rightarrow \alpha\ A'|\varepsilon$

$\therefore A \rightarrow \beta A'$ means $E \rightarrow TE'$

$A' \rightarrow \alpha\ A'|\varepsilon$ means $E' \rightarrow +TE'|\varepsilon$

Comparing $T \rightarrow T * F|F$ with $A \rightarrow A\alpha|\beta$

T	→	T	*F	|	F
A	→	A	α	|	β

$\therefore A = T, \alpha =* F, \beta = F$

∴ $A \rightarrow \beta A'$ means $T \rightarrow FT'$

$A \rightarrow \alpha A'|\varepsilon$ means $T' \rightarrow * FT'|\varepsilon$

Production $F \rightarrow (E)|id$ does not have any left recursion

∴ Combining productions 1, 2, 3, 4, 5, we get

$E \rightarrow TE'$

$E' \rightarrow +TE'|\ \varepsilon$

$T \rightarrow FT'$

$T \rightarrow * FT'|\varepsilon$

$F \rightarrow (E)|\ id$

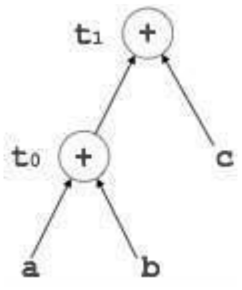
28. What is DAG?

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

- **Example:**

- $t_0 = a + b$
- $t_1 = t_0 + c$
- $d = t_0 + t_1$



29. **What is peephole optimization?**

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization: