[Total Nos. of Questions 6] **Roll No.** ☐

Pune Vidyarthi Griha's

COLLEGE OF ENGINEERING, NASHIK – 4.

B.E. (Computer Engineering)

COMPILER (2015 Pattern)

IN-SEM EXAM MARCH-2020

Time : 1 Hour ***[Max. Marks : 30]***

Instructions to the candidates:

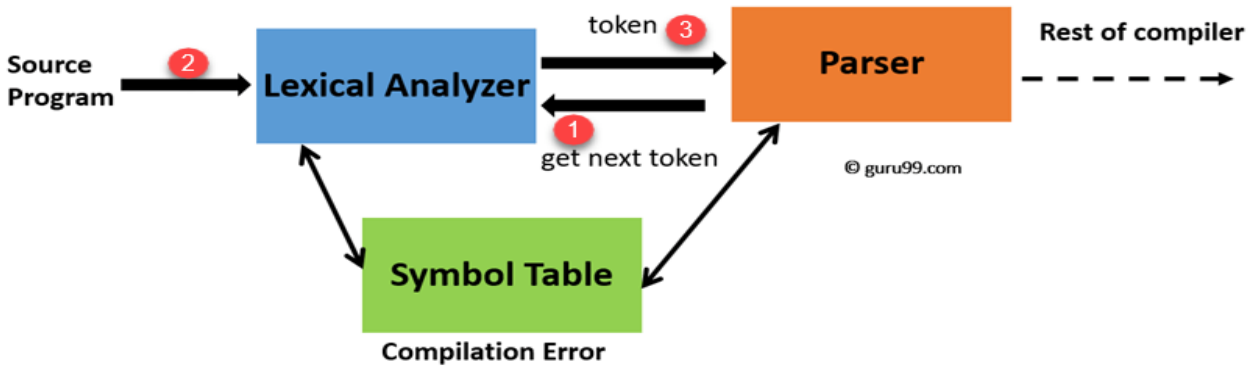
1) ***Solve Que. 1 or 2, Que.3 or 4, Que. 5 or 6.***
2) ***Neat diagrams must be drawn wherever necessary.***
3) ***Assume suitable data if necessary.***
4) ***Figures to the right indicate full marks.***

COMPILER MODEL ANSWER INSEM MARCH – 2020

Q. 1) a) Explain Role of Lexical Analyzer in details? **04 - M**

Ans :- **Role of Lexical Analyzer**

Interaction between lexical analyzer and parser



Lexical analyzer performs the following tasks:

- Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
- Enters the identified token into the symbol table.
- Strips out white spaces and comments from source program.
- Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.
- Expands the macros if it is found in the source program.

Tasks of lexical analyzer can be divided into two processes:

- Scanning: Performs reading of input characters, removal of white spaces and comments.
- Lexical Analysis: Produce tokens as the output.

Need of Lexical Analyzer

- Simplicity of design of compiler The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
- Compiler efficiency is improved Specialized buffering techniques for reading characters speed up the compiler process.

b) Write Lex specification for recognizing integer ,variable ,keyword & data types. 04-M
Ans :

//Main Program//

```
%{ # include<stdio.h>
FILE *fp;
int i=1;
%}
small [a-z]+
kwrd "void"
dat "int"|"float"|"char"|"string"
fun "main"
br "{"|"}"|"["|"]"|"("|")"
%%
{dat} {
      printf("\nLine : %d %s is DATATYPE",i,yytext);
     }
#include<{small}.h> {
                printf("\nLine : %d %s is HEADER FILE",i,yytext);
               }
{fun} {
      printf("\nLine : %d %s is FUNCTION",i,yytext);
     }
{br} {
      printf("\nLine : %d %s is BRACKETS",i,yytext);
     }
[a-zA-Z] {
      printf("\nLine : %d %s is VARIABLE",i,yytext);
       }
[0-9]*  {
      printf("\nLine : %d %s is NUMBER",i,yytext);
      }
[=] {
      printf("\nLine : %d %s is ASSIGNMENT OPERATER",i,yytext);
      }
```

```
[+|-|*|/] {
       printf("\nLine : %d %s is OPERATER",i,yytext);
        }
{kwrd} {
       printf("\nLine : %d %s is keyword",i,yytext);
      }
[\n] {
     printf("\n");
     i=i+1;
    }
%%

 main()
{
fp=fopen("prg.txt","r");
yyin=fp;
yylex();
fclose(fp);
}
```

//Input File To Given Program // prg.txt

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b,c;
a=5;
b=3;
c=a+b;
}
```

OUTPUT :

```
[engg10@Linux ~]$ lex token.l
[engg10@Linux ~]$ cc lex.yy.c -ll
[engg10@Linux ~]$ ./a.out
Line : 1 #include<stdio.h> is HEADER FILE
Line : 2 #include<conio.h> is HEADER FILE
Line : 3 void is keyword
Line : 3 main is FUNCTION
Line : 3 ( is BRACKETS
Line : 3 ) is BRACKETS
Line : 4 { is BRACKETS
Line : 5 int is DATATYPE
Line : 5 a is VARIABLE,
Line : 5 b is VARIABLE,
Line : 5 c is VARIABLE;

Line : 6 a is VARIABLE
Line : 6 = is ASSIGNMENT OPERATER
Line : 6 5 is NUMBER;
```

Line : 7 b is VARIABLE
Line : 7 = is ASSIGNMENT OPERATER
Line : 7 3 is NUMBER;
Line : 8 c is VARIABLE
Line : 8 = is ASSIGNMENT OPERATER
Line : 8 a is VARIABLE
Line : 8 + is OPERATER
Line : 8 b is VARIABLE;
Line : 9 } is BRACKETS

c) List Data structure used for implementation of Symbol Table? 02-M

Ans :

Following are commonly used data structure for implementing symbol table :-

List –

- In this method, an array is used to store names and associated information.
- A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from beginning of list till available pointer and if not found we get an error "use of undeclared name"
- While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. "Multiple defined name"
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- Advantage is that it takes minimum amount of space.

Linked List –

- This implementation is using linked list. A link field is added to each record.
- Searching of names is done in order pointed by link of link field.
- A pointer "First" is maintained to point to first record of symbol table.
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Hash Table –

- In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
- A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.
- To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.
- Insertion and lookup can be made very fast – O(1).
- Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

Binary Search Tree –

- Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follow the property of binary search tree.
- Insertion and lookup are O(log2 n) on average.

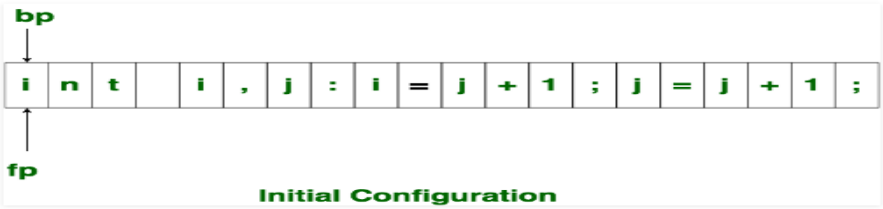
OR

Q 2) a) Explain input buffering of Lexical Analyzer ? 04-M

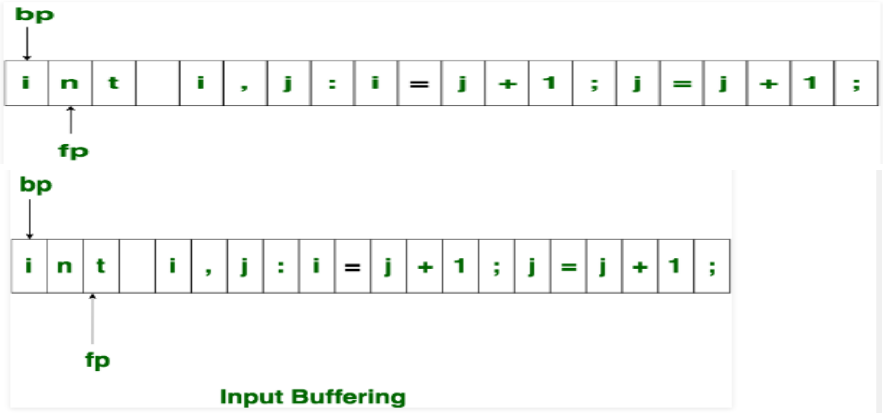
Ans :

Input Buffering

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(bp) and forward to keep track of the pointer of the input scanned.



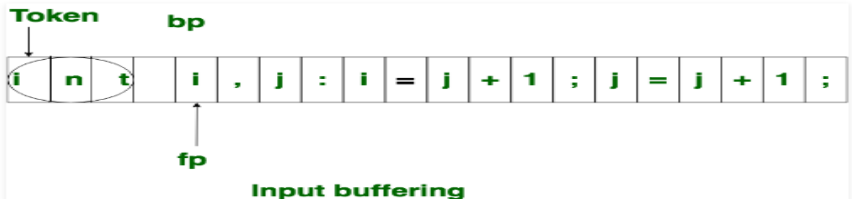
Initially both the pointers point to the first character of the input string as shown below



The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified.

The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.

The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below.



Two types : 1. Single buffering and 2. Double buffering

b) Write Lex specification for counting words , character and number of lines in given file. 06-M

Ans :

MAIN PROGRAM (lex file)

```
%{
int ch=0, bl=0, ln=0, wr=0;
 %}
 %%
[\n] {ln++;wr++;}
 [\t] {bl++;wr++;}
 [" "] {bl++;wr++;}
 [^\n\t] {ch++;}
%%
int main()
{
FILE *fp=fopen("abc.java","r");
yyin=fp;
 yylex();
printf("\nTotal No. of Characters are=%d\nTotal No. of Blank Spaces=%d\nTotal No. of Lines=%d\nTotal No. of Words=%d\n", ch, bl, ln, wr);
return 0;
}
```

Input File:

```
WELCOME TO PVGCOE,NASHIK
COLLEGE OF ENGINEERING GOOD MORNING GUYS
Hi Friends
Have a Wonderful Day Guys
Have a Great Day
```

**********OUTPUT**********

```
sl166@sl243-HP-Compaq-4000-Pro-SFF-PC:~$ vi a.l

sl166@sl243-HP-Compaq-4000-Pro-SFF-PC:~$ lex a.l

sl166@sl243-HP-Compaq-4000-Pro-SFF-PC:~$ gcc lex.yy.c -ll

sl166@sl243-HP-Compaq-4000-Pro-SFF-PC:~$ ./a.out

Total No. of Characters are=100

Total No. of Blank Spaces=14

Total No. of Lines=6

Total No. of Words=20
```

Q 3 a) Write YACC specification for expression grammar. 06-M

Ans :

yacc specification consists of a mandatory rules section, and optional sections for definitions and user subroutines.

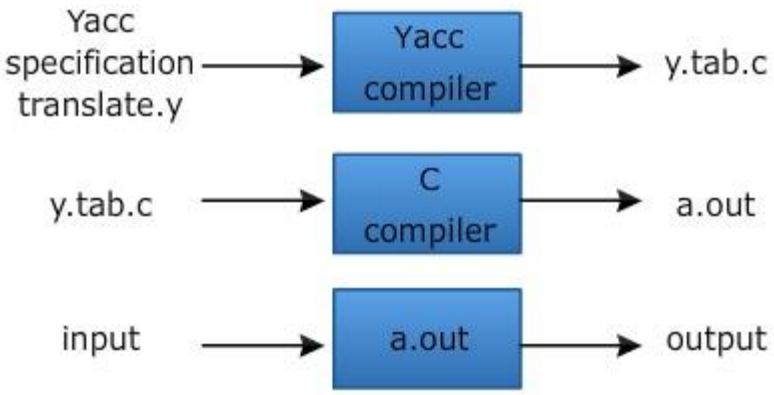
The declarations section for definitions, if present, must be the first section in the yacc program. The mandatory rules section follows the definitions; if there are no definitions, then the rules section is first. In both cases, the rules section must start with the delimiter %%. If there is a subroutines section, it follows the rules section and is separated from the rules by another %% delimiter. If there is no second %% delimiter, the rules section continues to the end of the file.

When all sections are present, a specification file has the format:

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```



Creating an Input/Output Translator with YACC

Expression grammar for YACC :

```
start: expr '\n'    {exit(1);}
     ;
expr:  expr '+' expr       {printf("+ ");}
    | expr '-' expr {printf("-");}
    | expr '*' expr {printf("*");}
    | expr '/' expr {printf("/");}

    | '(' expr ')'
    | DIGIT         {printf("NUM%d ",pos);}
    ;
```

lere is a grammar for assignment with arithmetic perations, e.g. `y = (2*x + 5)*x - 7;`

```
assignment => ID = expression ;
expression => expression + term
            | expression - term
            | term
term   =>     term * factor
            | term / factor
            | factor
factor =>     ( expression )
            | ID
            | NUMBER
```

b) Explain elimination of left recursion from the grammar. 04-M

Ans :

Definition :

A grammar is left-recursive if and only if there exists a nonterminal symbol A that can derive to a sentential form with itself as the leftmost symbol.Symbolically,

$A \rightarrow^{+} A\alpha$

where => + indicates the operation of making one or more substitutions, and α is any sequence of terminal and nonterminal symbols.

Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$$A \rightarrow A\alpha / \beta$$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' / \in$$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

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.

OR

Q 4. a) Calculate first and follow for following grammar. 06-M

Ans :

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

Then:

FIRST(E) = FIRST(T) = FIRST(F) = {(, **id**}
FIRST(E') = {+, ε}
FIRST(T') = {*, ε}

FOLLOW(E) = FOLLOW(E') = {) , $}
FOLLOW(T) = FOLLOW(T') = {+,), $}
FOLLOW(F) = {+, *,), $}

b) Explain shift reduce parsing on input id+id $ and consider foll. Grammar 04-M
Ans :

Consider the grammar:

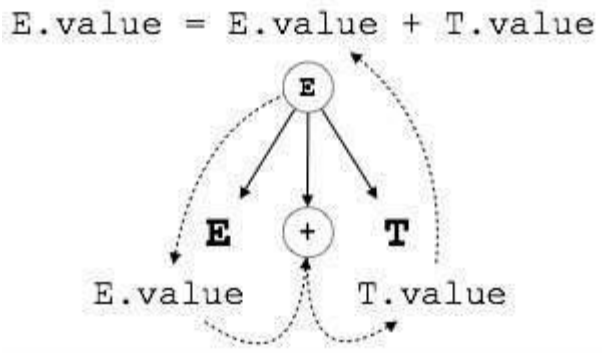
1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Stack	Input	Action
$	$id_1 + id_2$$	shift
$$id_1$	$+ id_2$$	reduce 6
$F	$+ id_2$$	reduce 4
$T	$+ id_2$$	reduce 2
$E	$+ id_2$$	shift
$E +	$id_2$$	shift
$E + id_2$		reduce 6
$E + F		reduce 4
$E + T		reduce 1
$E		accept

Q. 5 a) Explain S and L attributed Grammar. 04-M

Ans :

S-attributed SDT If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

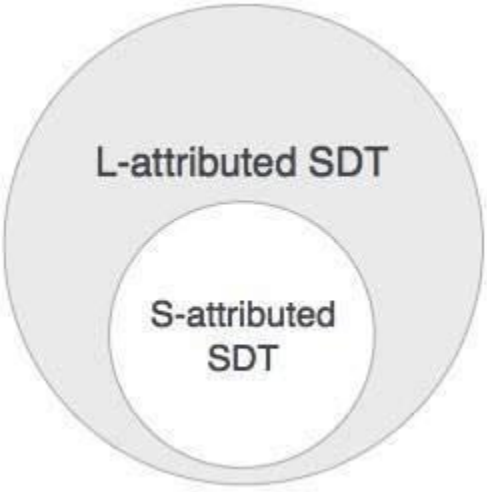


S-attributed SDT As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed SDT This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

S → ABC S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.



Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

L-attributed SDT We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

b) Explain Triple, Quadruple and Indirect triple with example. 06-M

Ans :

1. Quadruple –

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.

Advantage –

Easy to rearrange code for global optimization.

One can quickly access value of temporary variables using symbol table.

Disadvantage –

Contain lot of temporaries.

Temporary variable creation increases time and space complexity.

2. 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.

Disadvantage –

Temporaries are implicit and difficult to rearrange code.

It is difficult to optimize

3. 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.

Example – Consider expression a = b * – c + b * – c.

The three address code is:

```
t1 = uminus c

t2 = b * t1

t3 = uminus

t4 = b * t3

t5 = t2 + t4

a = t5
```

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

OR

Q. 6 a) Explain Syntax Directed Translation for Calculator. 04-M

Ans :

- Syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- It is a context free grammar with attributes and rules together which are associated with grammar symbols and productions respectively.
- The process of syntax directed translation is two-fold:
  - Construction of syntax tree and
  - Computing values of attributes at each node by visiting the nodes of syntax tree.

Syntax directed definition of simple desk calculator

Production	Semantic rules
L ---> E_n	*L.val = E.val*
E ---> E_1+ T	*E.val = E_1.val+ T.val*
E ---> T	*E.val = T.val*
T---> T_1*F	*T.val = T_i.val x F.val*
T ---> F	*T.val = F.val*
F ---> (E)	*F.val = E.val*
F ---> digit	*F.val = digit.lexval*

Syntax-directed definition-inherited attributes

roduction	Semantic Rules
D --->TL	*L.inh = T.type*
T ---> *int*	*T.type* =integer
T ---> *float*	*T.type = float*
L ---> L_1, **id**	L_1.inh = *L.inh* *addType (id.entry, Linh)*
L ---> **id**	*addType (id.entry, L.inh)*

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *inh*,

b) Write Syntax Directed Definition for flow of control statements. 06-M
(Ex . if-else, while any two)
Ans :

Production	Semantic Rule
$S \longmapsto$ **if** E **then** S_1	*E.true* := *newlabel*
	E.false := *S.next*
	S_1.*next* := *S.next*
	S.code := *E.code* | | *generate*(*E.true* '**:**') | | S_1.*code*
$S \longmapsto$ **if** E **then** S_1 **else** S_2	*E.true* := *newlabel*
	E.false := *newlabel*
	S_1.*next* := *S.next*
	S_2.*next* := *S.next*
	$code_1$:= *E.code* | | *generate*(*E.true* '**:**') | | S_1.*code*
	$code_2$:= *generate*('**goto**' *S.next*) | |
	$code_3$:= *generate*(*E.false* '**:**') | | S_2.*code*
	S.code := $code_1$ | | $code_2$ | | $code_3$
$S \longmapsto$ **while** E **repeat** S_1	*S.begin* := *newlabel*
	E.true := *newlabel*
	E.false := *S.next*
	S_1.*next* := *S.begin*
	$code_1$:= *generate*(*S.begin* '**:**') | | *E.code*
	$code_2$:= *generate*(*E.true* '**:**') | | S_1.*code*
	$code_3$:= *generate*('**goto**' *S.begin*)
	S.code := $code_1$ | | $code_2$ | | $code_3$

**************THE END**************