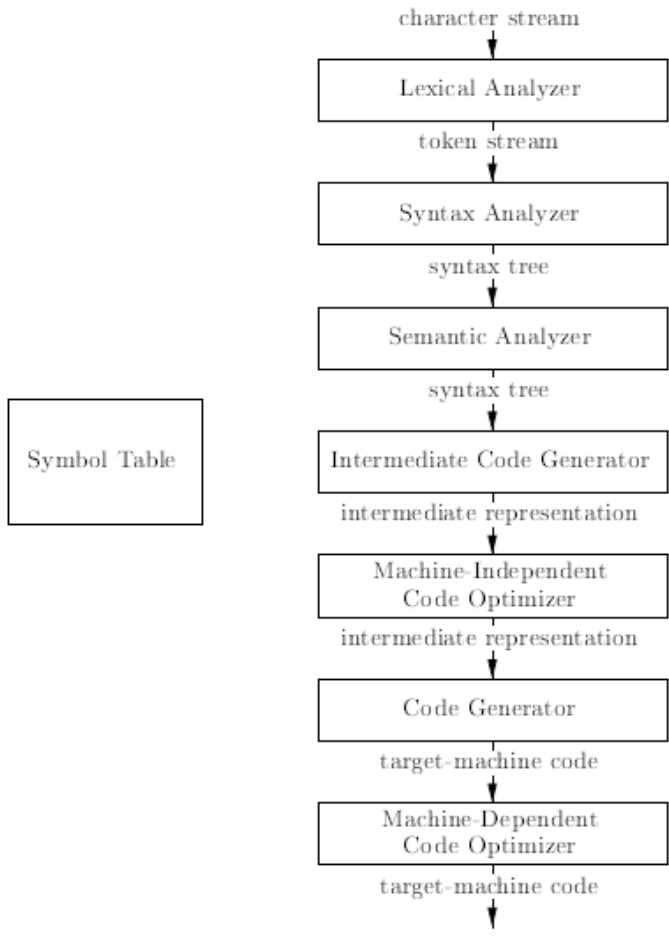


# CMR INSTITUTE OF TECHNOLOGY

Affiliated to VTU, Approved by AICTE, Accredited by NBA and NAAC with “A++” Grade  
 ITPL MAIN ROAD, BROOKFIELD, BENGALURU-560037, KARNATAKA, INDIA

## Department of Computer Science Engineering

### Answer Scheme & Model Solution- IAT1

Sub: System Software and Compilers		Sub Code: 18CS61	Sem/Branch: VI / CSE	Sections: A,B,C	
Question			MARKS	CO	RBT
1	i) Explain the various phases of a compiler with a neat diagram. ii) Show the translations for an assignment statement <u>sum =initial + value * 10</u> , clearly indicate the output of each phase.		10	CO2	L2
<b>Scheme Solution</b>			5+5		
	<p>(i)</p>  <pre> graph TD     A[character stream] --&gt; B[Lexical Analyzer]     B --&gt; C[token stream]     C --&gt; D[Syntax Analyzer]     D --&gt; E[syntax tree]     E --&gt; F[Semantic Analyzer]     F --&gt; G[syntax tree]     G --&gt; H[Intermediate Code Generator]     H --&gt; I[intermediate representation]     I --&gt; J[Machine-Independent Code Optimizer]     J --&gt; K[intermediate representation]     K --&gt; L[Code Generator]     L --&gt; M[target-machine code]     M --&gt; N[Machine-Dependent Code Optimizer]     N --&gt; O[target-machine code]   </pre> <p>Symbol Table</p> <p>Figure 1.6: Phases of a compiler</p>				

Lexical Analyzer: The 1<sup>st</sup> Phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Syntax Analyzer: The parser uses the 1<sup>st</sup> components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

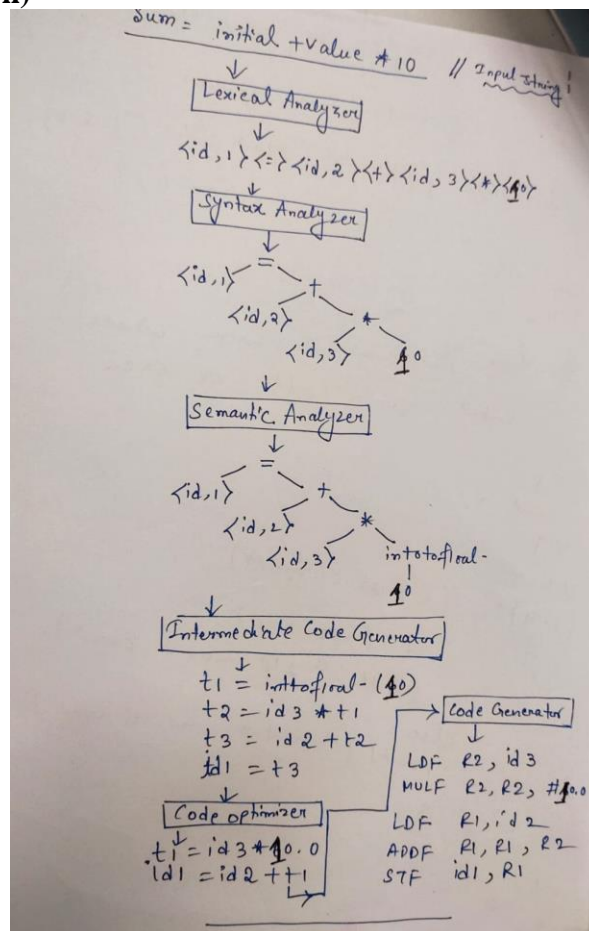
Semantic Analyzer: The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.

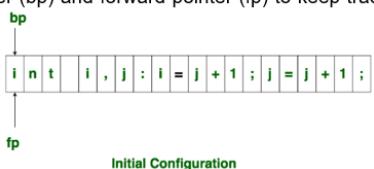
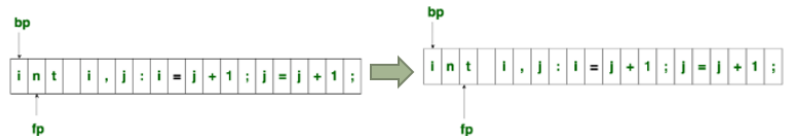
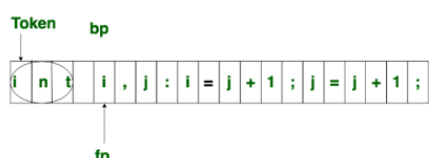
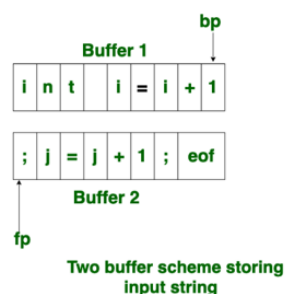
Intermediate Code generation: We consider an intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.

Optimizer: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

Code Generator: The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

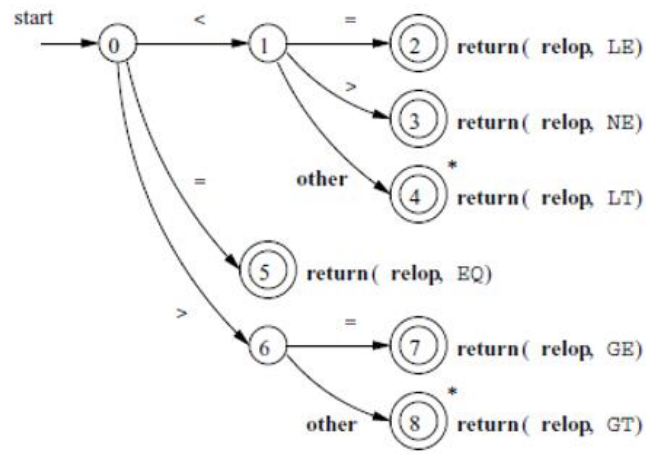
ii)



<b>Question</b>	2	Explain input buffering techniques in lexical analysis and justify why is it important. Also, explain the use of sentinels in recognizing the tokens.	10	CO2	L2
<b>Scheme</b>			10		
<b>Solution</b>	<p>To ensure that a right lexeme is found, often one or more characters have to be looked up beyond the next lexeme. there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for <b>id</b>. In C, single-character operators like -, =, or &lt; could also be the beginning of a two-character operator like &gt;, ==, or &lt;=.</p> <ul style="list-style-type: none"> <li>Thus, we shall introduce a <b>two-buffer scheme</b> that handles large lookaheads safely.</li> <li>We then consider an improvement involving <b>sentinels</b> that saves time checking for the ends of buffers.</li> </ul> <p>The lexical analyzer scans the input from left to right one character at a time. It uses two pointers lexemeBegin pointer (bp) and forward pointer (fp) to keep track of the pointer of the input scanned.</p>  <p>Initially both the pointers point to the first character of the input string as shown below</p>  <p>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.</p>  <p>To identify the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first <b>eof</b>, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second <b>eof</b> is obtained then it indicates of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified.</p> <p>This eof character introduced at the end is called <b>Sentinel</b> which is used to identify the end of buffer.</p>  <p style="text-align: center;"><b>Two buffer scheme storing input string</b></p>				

<b>Question</b>	3	i) Explain Token, Lexeme and Pattern with an example for each. ii) Write Lex Program to count words, characters, lines in a given input file	10	CO3	L2
<b>Scheme</b>			6+4		
<b>Solution</b>		<p>i) A <b>token</b> is a pair a token name and an optional token value ex: keyword, identifier.-if else and num1,num2 A <b>pattern</b> is a description of the form that the lexemes of a token may take Ex: identifier: ([a-z][A-Z]) ([a-z][A-Z][0-9])* A <b>lexeme</b> is a sequence of characters in the source program that matches the pattern for a token. ex: printf("total = %d\n", score); both <b>printf</b> and <b>score</b> are lexemes matching the pattern for token <b>id</b>, and "<b>Total = %d\n</b>" is a lexeme matching <b>literal</b>.</p> <p>ii)</p> <pre> %{ unsigned charCount = 0, wordCount = 0, lineCount = 0; %}  word [^ \t\n]+ eol \n  %% (word)      { wordCount++; charCount += yyleng; } (eol) { charCount++; lineCount++; } .          charCount++; main(argc,argv) int argc; char **argv; {     if (argc &gt; 1) {         FILE *file;          file = fopen(argv[1], "r");         if (!file) {             fprintf(stderr, "could not open %s\n", argv[1]);             exit(1);         }         yyin = file;     }     yylex();     printf("%d %d %d\n", charCount, wordCount, lineCount);     return 0; } </pre>			
<b>Question</b>	4	Construct transition diagram for recognizing relation operators and also demonstrate the program segment to implement it.	10	CO2	L3
<b>Scheme</b>			5+5		

**Solution**



```

TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
                    ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}

```

<p><b>Question</b></p>	<p>5</p>	<p>i) Write a lex program to recognize the tokens (keywords {if, then, else}, number, relop, id) and return the tokens considering attribute values.                      ii) Demonstrate the use of following Meta Characters with suitable example:                      + , \$ , ^ , { } , \d</p>	<p>10</p>	<p>CO3</p>	<p>L3</p>
<p><b>Scheme</b></p>			<p>5+5</p>		
<p><b>Solution</b></p>	<p>i)</p>				

```

%{
/* definitions of manifest constants
   LT, LE, EQ, NE, GT, GE,
   IF, THEN, ELSE, ID, NUMBER, RELOP */
}%
/* regular definitions */
delim  [ \t\n]
ws     {delim}+
letter [A-Za-z]
digit  [0-9]
id     {letter} ({letter}{digit})*
number {digit}+ (\. {digit}+)? (E [+]?{digit}+)?

%%
{ws}  { /* no action and no return */}
if    {return(IF);}
then  {return(THEN);}
else  {return(ELSE);}
{id}  {yylval =(int) installID() ; return(ID);}
{number} {yylval =(int) installNum() ;return(NUMBER) ; }
"<"   {yylval = LT; return(RELOP);}
"<="  {yylval = LE; return(RELOP);}
"="   {yylval = EQ; return(RELOP) ;}
"<>"  {yylval = NE; return(RELOP);}

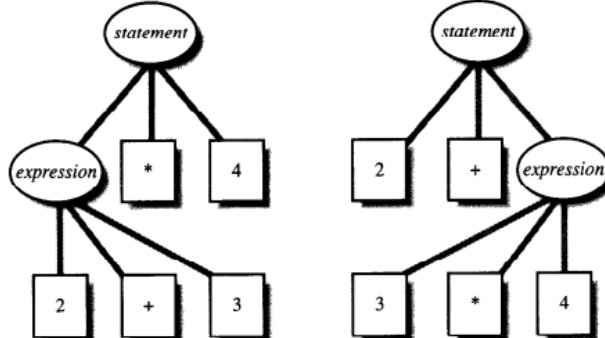
">"   {yylval = GT; return(RELOP);}
">="  {yylval = GE; return(RELOP);}
%%

//auxiliary functions
int installID() {
    /* function to install the lexeme,
       whose first character is pointed to by yytext,
       a d whose length is yyleng,
       into the symbol table and
       return a pointer thereto */
}
int installNum() {
    /* similar to installID, but puts numerical
       constants into a separate table */
}
main()
{
    yylex();
}

```

ii)

- ^ Matches the beginning of input.
- \$ Matches the end of input.
- + Matches the preceding character one or more times.  
For example, zo+ matches zoo but not z.

	<p><b>{n}</b> n is a non-negative integer. Matches exactly n times. For example, o{2} does not match the o in Bob, but matches the first two os in foood.</p> <p><b>\d</b> Matches a digit character.</p>			
<p><b>Question</b></p>	<p>6</p> <p>i) With an example, demonstrate ambiguous grammar and show how can it be overcome. ii) Write a YACC program to check whether the given arithmetic expression is valid or not</p>	<p>10</p>	<p>CO3</p>	<p>L3</p>
<p><b>Scheme</b></p>		<p>5+5</p>		
<p><b>Solution</b></p>	<p>i)</p> <pre> expression: expression '+' expression { \$\$ = \$1 + \$3; }   expression '-' expression { \$\$ = \$1 - \$3; }   expression '*' expression { \$\$ = \$1 * \$3; }   expression '/' expression   { if(\$3 == 0)     yyerror("divide by zero");     else       \$\$ = \$1 / \$3;   }   '-' expression { \$\$ = -\$2; }   '(' expression ')' { \$\$ = \$2; }   NUMBER { \$\$ = \$1; } ; </pre> <p><b>this grammar has a problem: it is extremely ambiguous.</b> For example, the input 2+3*4 might mean (2+3)*4 or 2+(3*4), and the input 3-4-5-6 might mean 3-(4-(5-6)) or (3-4)-(5-6) or any of a lot of other possibilities. Following Figure shows the two possible parses for 2+3*4.</p>  <p>The problem can be solved by specifying precedence and associativity of the operators.</p> <p><u>Explicitly Specifying Precedence and Associativity:</u></p> <pre> %left '+' '-' %left '*' '/' %nonassoc LIMTUS </pre>			

Implicitly Specifying Precedence and Associativity:

```
expression: expression '+' mulexp
           | expression '-' mulexp
           | mulexp
           ;

mulexp:    mulexp '**' primary
           | mulexp '/' primary
           | primary
           ;

primary:   '(' expression ')'
           | '-' primary
           | NUMBER
           ;
```

ii)

Lex Part

```
%{
#include<stdio.h>
#include "y.tab.h"
extern yylval;
}%
%%
[0-9]+ {yylval=atoi(yytext);return num;}
[+\-\\*√] {return yytext[0];}
[] {return yytext[0];}
[(] {return yytext[0];}
. {;}
\n {return 0;}
%%
```

Yacc part:

```
%{
#include<stdio.h>
#include<stdlib.h>
}%
%token num
%left '+' '-'
%left '*' '/'
```



```

%%
input:exp {printf("%d\n",$$);exit(0);}
exp:  exp'+exp {$$=$1+$3;}
      |exp'-exp {$$=$1-$3;}
      |exp'*exp {$$=$1*$3;}
      |exp'/exp { if($3==0){printf("Divide by Zero.
Invalid expression.\n");exit(0);}
      else $$=$1/$3;}
      |('exp'){$$=$2;}
      |num{$$=$1;};

%%

int yyerror()
{
    printf("Error. Invalid Expression.\n");
    exit(0);
}

int main()
{
    printf("Enter an expression:\n");
    yyparse();
}

```