Course Code	Course Title						Core/ Elective
PC 631 CS	**COMPILER DESIGN LAB**						**CORE**
Prerequisite	Contact Hours Per Week				CIE	SEE	Credits
	L	T	D	P			
-	-	-	-	2	25	50	2

Course Objectives

- To learn usage of tools LEX, YAAC
- To develop a code generator
- To implement different code optimization schemes

Course Outcomes

- Generate scanner and parser from formal specification.
- Generate top down and bottom up parsing tables using Predictive parsing, SLR and LR Parsing techniques.
- Apply the knowledge of YACC to syntax directed translations for generating intermediate code – 3 address code.
- Build a code generator using different intermediate codes and optimize the target code.

List of Experiments to be performed:

1. Sample programs using LEX.
2. Scanner Generation using LEX.
3. Elimination of Left Recursion in a grammar.
4. Left Factoring a grammar.
5. Top down parsers.
6. Bottom up parsers.
7. Parser Generation using YACC.
8. Intermediate Code Generation.
9. Target Code Generation.
10. Code optimization.

1.1 Lex program to count the number of words, characters, blank spaces and lines

Procedure:

1. Create a lex specification file to recognize words, characters, blank spaces and lines
2. Compile it by using LEX compiler to get 'C' file
3. Now compile the c file using C compiler to get executable file
4. Run the executable file to get the desired output by providing necessary input

Program:

```
%{

int c=0,w=0,l=0,s=0;

%}

%%

[\n]  l++;

[' '\n\t]  s++;

[^' '\t\n]+  w++; c+=yyleng;

%%

int main(int argc, char *argv[])

{

if(argc==2)

{

yyin=fopen(argv[1],"r");

yylex();

printf("\nNUMBER OF SPACES = %d",s);

printf("\nCHARACTER=%d",c);

printf("\nLINES=%d",l);
```

```
printf("\nWORD=%d\n",w);

}

else
printf("ERROR");
}
```

Input File:

Hello how are you

Output:

```
 lex filename.l
cc lex.yy.c -ll
./a.out in.txt
```

1.2 LEX program to identify REAL PRECISION of the given number

Procedure:

1. Create a lex specification file to recognize single line and multiple comment statements
2. Compile it by using LEX compiler to get 'C' file
3. Now compile the c file using C compiler to get executable file
4. Run the executable file to get the desired output by providing necessary input

Program:

```
%{
/*Program to identify a integer/float precision*/
%}
integer ([0-9]+)
float ([0-9]+\.[0-9]+)|([+|-]?[0-9]+\.[0-9]*[e|E][+|-][0-9]*)

%%
{integer}
printf("\n %s is an integer.\n",yytext);
{float} printf("\n %s is a floating number.\n",yytext);
%%

main()
{
yylex();
}
int yywrap()
{
return 1;
}
```

Output:

```
lex filename.l
cc lex.yy.c -ll
./a.out
```

*Pass the number

2 is an integer

2.3
2.3 is a floating number

2.1 Scanner generation using LEX

Concepts:

LEX Program Concepts:

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in C .

A **token** is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is

called **tokenization**, and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

Consider this expression in the C programming language:

sum=3+2;

Tokenized in the following table:

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Integer literal
+	Addition operator
2	Integer literal
;	End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens.

This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

The structure of a Lex file is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

The following LEX Program is to recognize tokens given as C source code statements and return token values.

```
/* LEX Program to recognize tokens */
%{
#define LT 256
#define LE 257
#define EQ 258
#define NE 259
#define GT 260
#define GE 261
#define RELOP 262
#define ID 263
#define NUM 264
#define IF 265
#define THEN 266
#define ELSE 267
int attribute;
%}
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws} {}
if { return(IF); }
then { return(THEN); }
else { return(ELSE); }
{id} { return(ID); }
{num} { return(NUM); }
"<" { attribute=LT;return(RELOP); }
"<=" { attribute=LE;return(RELOP); }
"<>" { attribute=NE;return(RELOP); }
"=" { attribute=EQ;return(RELOP); }
">" { attribute=GT;return(RELOP); }
">=" { attribute=GE;return(RELOP); }
%%
int yywrap(){
return 1;
```

```
}
int main()
{
int token;
while(token=yylex()){
printf("<%d,",token);
switch(token){
case ID:case NUM:
printf("%s>\n",yytext);
break;
case RELOP:
printf("%d>\n",attribute);
break;
default:
printf(")\n");
break;
}
}
return 0;
}
```

OUTPUT:

```
$ lex filename.l
$ cc lex.yy.c -ll
$ ./a.out
if a>b then a else b
<265>
<263,a>
<262,260>
<263,b>
<266>
<263,a>
<267>
<263,b>
```

2.2. Implementation Of Lex Analyzer Tool.

Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text. Lex is an acronym that stands for "lexical analyzer generator." It is intended primarily for Unix-based systems. The code for Lex was originally developed by Eric Schmidt and Mike Lesk.

Lex can perform simple transformations by itself but its main purpose is to facilitate lexical analysis, the processing of character sequences such as source code to produce symbol sequences called tokens for use as input to other programs such as parsers. Lex can be used with a parser generator to perform lexical analysis. It is easy, for example, to interface Lex and Yacc, an open source program that generates code for the parser in the C programming language.

The general format of Lex source is:

{definitions}
%%
{rules}
%%
{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column contains

actions, program fragments to be executed when the expressions are recognized. Thus an individual rule might appear

integer printf("found keyword INT");

to look for the string integer in the input stream and print the message ``found keyword INT" whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces. As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

colour printf("color");
mechanise printf("mechanize");
petrol printf("gas");

would be a start. These rules are not quite enough, since theword petroleum would become gaseum; a way of dealing with thiswill be described later.

Lex Regular Expressions.

A regular expression specifies a set of strings to be matched. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters; thus the regular expression

integer

matches the string integer wherever it appears and the expression

a57D

looks for the string a57D.

Operators. The operator characters are

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters. Thus

xyz"++"

matches the string xyz++ when it appears. Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

"xyz++"

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

An operator character may also be turned into a text character by preceding it with \ as in

xyz\+\+

which is another, less readable, equivalent of the above expressions. Another use of the quoting mechanism is to get a blank into an expression; normally, as explained above, blanks or tabs end a rule. Any blank character not contained within [] (see below) must be quoted. Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \

itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.

Character classes. Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges. For example,

[a-z0-9<>_]

indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order. Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. (E.g., [0-z] in ASCII is many more characters than it is in EBCDIC). If it is desired to include the character - in a character class, it should be first or last; thus

[-+0-9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus

[^abc]

matches all characters except a, b, or c, including all special or

control characters; or

[^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

Arbitrary character. To match almost any character, the operator character

.

is the class of all characters except newline. Escaping into octal is possible although non-portable:

[\40-\176]

matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde).

Optional expressions. The operator ? indicates an optional element of an expression. Thus

ab?c

matches either ac or abc.

Repeated expressions. Repetitions of classes are indicated by the operators * and +.

a*

is any number of consecutive a characters, including zero; while

a+

is one or more instances of a. For example,

[a-z]+

is all strings of lower case letters. And

[A-Za-z][A-Za-z0-9]*

indicates all alphanumeric strings with a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

Alternation and Grouping. The operator | indicates alternation:

(ab|cd)

matches either ab or cd. Note that parentheses are used for grouping, although they are not necessary on the outside level;

ab|cd

would have sufficed. Parentheses can be used for more complex expressions:

(ab|cd+)?(ef)*

matches such strings as abefef, efefef, cdef, or cddd; but not abc, abcd, or abcdef.

Echo-→ matching new line

Yytext--→ matching string

```
/* Implementation of LEX Tool */
%{
/*lex program to recognize a  C program*/
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*


%%

#.* {printf("\n %s is a PREPROCESSOR DIRECTIVE",yytext);}
int |
float |
char |
double |
while |
for |
do |
if |
break |
continue |
void |
switch |
case |
long |
struct |
const |
typedef |
return |
else |   /*…..*/
goto {printf("\n\t %s is a KEYWORD",yytext);}
"/*" {COMMENT=1; printf("\n\n\t %s is a COMMENT\n",yytext);}
"*/" {COMMENT=0; printf("\n\n\t %s is a COMMENT\n",yytext);}
{identifier}\( {if(!COMMENT)printf("\n\n FUNCTION\n\t %s",yytext);}
\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
\} {if(!COMMENT)printf("\n BLOCK ENDS");}
{identifier}(\[[0-9]*\])? {if(!COMMENT)printf("\n %s
IDENTIFIER",yytext);}
\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
[0-9]+ {if(!COMMENT)printf("\n\t %s is a NUMBER",yytext);}
```

```
\)(\;)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\(  ECHO;
= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT
OPERATOR",yytext);}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT)printf("\n\t %s is a RELATIONAL
OPERATOR",yytext);}
%%

int main(int argc,char** argv)
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("could not open %s\n",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
:r // input exhausted
{return 0; // no more input to process
}
```

Var.c

```
/* This is LEx Tool Program */
#include<stdio.h>
main()
{
int a,b;
}
```

OUTPUT:

```
$ lex lextool.l
$ cc lex.yy.c –ll
$ ./a.out  var.c

        /* is a COMMENT


      */ is a COMMENT


#include<stdio.h> is a PREPROCESSOR DIRECTIVE


FUNCTION
        main(
       )


BLOCK BEGINS

        int is a KEYWORD
a IDENTIFIER,
b IDENTIFIER;

BLOCK ENDS
```

3. Elimination of Left Recursion in a grammar.

```
#include<stdio.h>
#include<string.h>
void main()
{
char input[100], l[50],r[50],temp[10],tempprod[20],productions[25][50];
int i=0,j=0,flag=0,consumed=0;
 printf("Enter the Productions:");
   scanf(" %ls->%s", l, r);
    printf(" %s", r);
while(sscanf(r+consumed, " % [^l] s", temp) == 1 &&consumed<=strlen(r))
{
   if(temp[0] == l[0])
     {
        flag = 1;
   sprintf(productions[i++], "%s->%s%s '\0", l,temp+1,1);
       }
else
      sprintf(productions[i++], "%s->%s%s '\0",l, temp,1);
      consumed += strlen(temp)+1;
}
if(flag==1)
{
  sprintf(productions[i++], "%s->€ \0", 1);
  printf("the productions after eliminating left recursion are:\n");
for(j=0;j<i;j++)
 printf("%s \n ", productions[j]);
}
 else
  printf(" The Given Grammar has no Left Recursion");
}
```

OUTPUT:

```
Enter the Productions:
E->E+T
```

The productions after eliminating Left Recursion are:E->+TE’
E->
Enter the Productions:
T->T*F
The productions after eliminating Left Recursion are: T-> *FT’
T->
Enter the Productions:
F->id
The Given Grammar has no Left Recursion

5.1 Top down parsers.

Implementation Of Recursive Descent Parsing.
Concepts:
A *recursive descent parser* is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools. It is a simple and effective technique, but is not as powerful as some of the shift-reduce parsers -- not the one presented in class, but fancier similar ones called *LR parsers*.

This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

Perhaps the hardest part of a recursive descent parser is the scanning: repeatedly fetching the next token from the scanner. It is tricky to decide when to scan, and the parser doesn't work at all if there is an extra scan or a missing scan.

Initial Example:

- Consider the grammar used before for simple arithmetic expressions

P ---> E
E ---> E + T | E - T | T
T ---> T * S | T / S | S
S ---> F ^ S | F
F ---> (E) | char

- The above grammar won't work for recursive descent because of the left recursion in the second and third rules. (The recursive function for **E** would immediately call **E** recursively, resulting in an indefinite recursive regression.)

  In order to eliminate left recursion, one simple method is to introduce new notation: curley brackets, where **{xx}** means "zero or more repetitions of **xx**", and parentheses **()** used for grouping, along with the or-symbol: |. Because of the many metasymbols, it is a good idea to enclose all terminals in single quotes. Also put a **'$'** at the end. The resulting grammar looks as follows:

  P ---> E '$'
  E ---> T {('+'|'-') T}
  T ---> S {('*'|'/') S}
  S ---> F '^' S | F
  F ---> '(' E ')' | char

  Now the grammar is suitable for creation of a recursive descent parser. Notice that this is a *different* grammar that describes the *same* language, that is the same sentences or strings of terminal symbols. A given sentence will have a similar parse tree to one given by the previous grammar, but not necessarily the same parse tree.

One could alter the first grammar in other ways to make it work for recursive descent. For example, one could write the rule for **E** as:

E ---> T '+' E | T

This eliminates the left recursion, and leaves the language the same, but it changes the *semantics* of the language. With this change, the operator '+' would *associate from right to left*, instead of from left to right, so this method is not acceptable.

/* Program to Implement Recursive Descent Parsing */

```
#include<stdio.h>
//#include<conio.h>
#include<string.h>
char input[100];
int i,l;
int main()
{
printf("recursive decent parsing for the grammar");
printf("\n E->TEP|\nEP->+TEP|@|\nT->FTP|\nTP->*FTP|@|\nF-
>(E)|ID\n");
printf("enter the string to check:");
scanf("%s",input);
if(E()){
if(input[i]=='$')
printf("\n string is accepted\n");
else
printf("\n string is not accepted\n");
}
}
E(){
if(T()){
if(EP())
return(1);
else
return(0);
}
else
return(0);
}
EP(){
if(input[i]=='+'){
i++;
if(T()){
if(EP())
return(1);
else
return(0);
```

```
}
else
return(1);
}
}
T(){
if(F()){
if(TP())
return(1);
else
return(0);
}
else
return(0);
printf("String is not accpeted\n");
}
TP(){
if(input[i]=='*'){
i++;
if(F()){
if(TP())
return(1);
else
return(0);
}
else
return(0);
printf("The string is not accepted\n");
}
else
return(1);
}
F(){
if(input[i]=='('){
i++;
if(E()){
if(input[i]==')'){
i++;
return(1);
}
```

```
else
{
return(0);
printf("String is not accepted\n");
}
}
else
return(0);
}
else
if(input[i]>='a'&&input[i]<='z'||input[i]>='A'&&input[i]<='Z')
{
i++;
return(1);
}
else
return(0);
}
```

OUTPUT:

```
$ cc rdp.c
$ ./a.out
recursive decent parsing for the grammar
 E->TEP|
EP->+TEP|@|
T->FTP|
TP->*FTP|@|
F->(E)|ID
enter the string to check:(i+i)*i

 string is accepted
```

5.2 Implementation Of Predictive Parsing.

Concepts:

Predictive parsing can also be accomplished using a *predictive parsing table* and a stack. It is sometimes called non-recursive predictive parsing. The idea is that we construct a table *M*[*X*, *token*] which indicates which production to use if the top of the stack is a nonterminal *X* and the current token is equal to *token*; in that case we pop *X* from the stack and we push all the rhs symbols of the production *M*[*X*, *token*] in reverse order. We use a special symbol $ to denote the end of file. Let *S* be the start symbol. Here is the parsing algorithm:

```
push(S);
read_next_token();
repeat
  X = pop();
  if (X is a terminal or '$')
     if (X == current_token)
        read_next_token();
     else error();
  else if (M[X,current_token] == "X ::= Y1 Y2 ... Yk")
     {  push(Yk);
        ...
        push(Y1);
     }
  else error();
until X == '$';
```

Now the question is how to construct the parsing table. We need first to derive the *FIRST*[*a*] for some symbol sequence *a* and the *FOLLOW*[*X*] for some nonterminal *X*. In few words, *FIRST*[*a*] is the set of terminals *t* that result after a number of derivations on *a* (ie, *a* = > ... = > *tb* for some *b*). For example, *FIRST*[3 + *E*] = {3} since 3 is the first terminal. To find *FIRST*[*E* + *T*], we need to

apply one or more derivations to *E* until we get a terminal at the beginning. If *E* can be reduced to the empty sequence, then the *FIRST*[*E* + *T*] must also contain the *FIRST*[+ *T*] = { + }. The *FOLLOW*[*X*] is the set of all terminals that follow *X* in any legal derivation. To find the *FOLLOW*[*X*], we need find all productions *Z* : : = *aXb* in which *X* appears at the rhs. Then the *FIRST*[*b*] must be part of the *FOLLOW*[*X*]. If *b* is empty, then the *FOLLOW*[*Z*] must be part of the *FOLLOW*[*X*].

Consider our CFG G1:

```
1)  E ::= T E' $
2)  E' ::= + T E'
3)       | - T E'
4)       |
5)  T ::= F T'
6)  T' ::= * F T'
7)       | / F T'
8)       |
9)  F ::= num
10)      | id
```

FIRST[F] is of course {num,id}. This means that FIRST[T]=FIRST[F]={num,id}. In addition, FIRST[E]=FIRST[T]={num,id}. Similarly, FIRST[T'] is {*,/} and FIRST[E'] is {+,-}.

The FOLLOW of E is {$} since there is no production that has E at the rhs. For E', rules 1, 2, and 3 have E' at the rhs. This means that the FOLLOW[E'] must contain both the FOLLOW[E] and the FOLLOW[E']. The first one is {$} while the latter is ignored since we are trying to find E'. Similarly, to find FOLLOW[T], we find the rules that have T at the rhs: rules 1, 2, and 3. Then FOLLOW[T] must include FIRST[E'] and, since *E'* can be reduced to the empty sequence, it must include FOLLOW[E'] too (ie, {$}). That is, FOLLOW[T]={+,-,$}.

Similarly, FOLLOW[T']=FOLLOW[T]={+,-,$} from rule 5.
The FOLLOW[F] is equal
to FIRST[T']={*,/} plusFOLLOW[T'] and plus FOLLOW[T] from rules 5, 6, and 7, since T' can be reduced to the empty sequence.

To summarize, we have:

	FIRST	FOLLOW
E	{num,id}	{$}
E'	{+,-}	{$}
T	{num,id}	{+,-,$}
T'	{*,/}	{+,-,$}
F	{num,id}	{+,-,*,/,$}

Now, given the above table, we can easily construct the parsing table. For each $t \overset{\in}{\_\_} FIRST[a]$, add $X : : = a$ to $M[X, t]$. If a can be reduced to the empty sequence, then for each $t \overset{\in}{\_\_} FOLLOW[X]$, add $X : : = a$ to $M[X,t]$.

For example, the parsing table of the grammar G1 is:

	num	id	+	-	*	/	$
E	1	1					
E'			2	3			4
T	5	5					
T'			8	8	6	7	8
F	9	10					

where the numbers are production numbers. For example, consider the eighth production *T'* : : = . Since FOLLOW[T']={+,-,$}, we add 8 to M[T',+], M[T',-], M[T',$].

A grammar is called LL(1) if each element of the parsing table of the grammar has at most one production element. (The first L in LL(1) means that we read the input from left to right, the second L means that it uses left-most derivations only, and the number 1 means that we need to look one token only ahead from the input.) Thus G1 is LL(1). If we have multiple entries in *M*, the grammar is not LL(1).

We will parse now the string x-2*y$ using the above parse table:

Stack	current_token	Rule
E	x	M[E,id] = 1 (using E ::= T E' $)
$ E' T	x	M[T,id] = 5 (using T ::= F T')
$ E' T' F	x	M[F,id] = 10 (using F ::= id)
$ E' T' id	x	read_next_token
$ E' T'	-	M[T',-] = 8 (using T' ::=)
$ E'	-	M[E',-] = 3 (using E' ::= - T E')
$ E' T -	-	read_next_token
$ E' T	2	M[T,num] = 5 (using T ::= F T')
$ E' T' F	2	M[F,num] = 9 (using F ::= num)
$ E' T' num	2	read_next_token
$ E' T'	*	M[T',*] = 6 (using T' ::= * F T')
$ E' T' F *	*	read_next_token
$ E' T' F	y	M[F,id] = 10 (using F ::= id)
$ E' T' id	y	read_next_token
$ E' T'	$	M[T',$] = 8 (using T' ::=)
$ E'	$	M[E',$] = 4 (using E' ::=)
$	$	stop (accept)

As another example, consider the following grammar:

G ::= S $
S ::= (L)
| a
L ::= L , S
| S

After left recursion elimination, it becomes

0) G := S $
1) S ::= (L)
2) S ::= a
3) L ::= S L'
4) L' ::= , S L'
5) L' ::=

The first/follow tables are:

	FIRST	FOLLOW
G	(a	
S	(a	,) $
L	(a	)
L'	,	)

which are used to create the parsing table:

	(	)	a	,	$
G	0		0		
S	1		2		
L	3		3		
L'		5		4	

```
/* Program to Implement Predictive Parsing */
#include<stdio.h>
#include<string.h>
int spt=0,ipt=0;
char s[20],ip[15];
 char *m[5][6]={{“TG”,”\0”,”\0”,”TG”,”\0”,”\0”},
                {“\0”,”+TG”,”\0”,”\0”,”e”,”e”},
                {“FH”,”\0”,”\0”,”FH”,”\0”,”\0”},
                {“\0”,”e”,”*FH”,”\0”,”e”,”e”},
                {“i”,”\0”,”\0”,”(E)”,”\0”,”\0”}};
char nt[5]={'E','G','T','H','F'};
char t[6]={'i','+','*','(',')','$'};
int nti(char c)
{
int i;
for(i=0;i<5;i++){
if(nt[i]==c)
return(i);
}
return(6);
}
int ti(char c)
{
int i;
for(i=0;i<6;i++)
{
if(t[i]==c)
return(i);
}
return(7);
}
main()
{
char prod[4],temp[4];
int l,k,j;
printf("enter input string:");
scanf("%s",ip);
strcat(ip,"$");
s[0]='$';
s[1]='E';
```

```
s[2]='\0';
spt=1;
while(1)
{
if(ip[ipt]=='$'&&s[spt]=='$')
break;
if(ti(s[spt])<5||s[spt]=='$')
{
if(s[spt]==ip[ipt])
{
spt--;
ipt++;
}
else
error();
}
else if(nti(s[spt]<6)){
strcpy(prod,m[nti(s[spt])][ti(ip[ipt])]);
if(prod=='\0')
error();
l=strlen(prod);
for(k=l-1,j=0;k>=0&&j<=l;k--,j++)
temp[j]=prod[k];
for(k=0;k<l;k++)
prod[k]=temp[k];
s[spt--]='\0';
strcat(s,prod);
spt=spt+l;
if(s[spt]=='e')
s[spt--]='\0';
}
else
error();
}
if(s[spt]=='$'&&ip[ipt]=='$')
printf("\n input is parsed\n");
else
error();
return 0;
}
```

```
error()
{
printf("input is not parsed\n");
exit(1);
return 0;
}
```

OUTPUT:

```
$ cc preparsing.c
$ ./a.out
enter input string:i+i*i$

 input is parsed
```

5.3 Implementation Of LL(1) Parsing.

Concepts :

In computer science, an **LL parser** is a top-down parser for a subset of the context-free grammars. It parses the input from **L**eft to right, and constructs a **L**eftmost derivation of the sentence (hence LL, compared with LR parser that constructs a rightmost derivation). The class of grammars which are parsable in this way is known as the *LL grammars*.

An LL parser is called an LL(*k*) parser if it uses *k* tokens of lookahead when parsing a sentence. If such a parser exists for a certain grammar and it can parse sentences of this grammar withoutbacktracking then it is called an LL(*k*) grammar. A language that has an LL(k) grammar is known as an LL(k) language. There are LL(k+n) languages that are not LL(k) languages. A corollary of this is that not all context-free languages are LL(k) languages.

LL(1) grammars are very popular because the corresponding LL parsers only need to look at the next token to make their parsing decisions. Languages based on grammars with a high value of *k* have traditionally been considered to be difficult to parse, although this is less true now given the availability and widespread use of parser generators supporting LL(*k*) grammars for arbitrary *k*.

An LL parser is called an LL(*) parser if it is not restricted to a finite *k* tokens of lookahead, but can make parsing decisions by recognizing whether the following tokens belong to a regular language (for example by use of a Deterministic Finite Automaton).

The parser works on strings from a particular context-free grammar.

The parser consists of an *input buffer*, holding the input string (built from the grammar)

- a *stack* on which to store the terminals and non-terminals from the grammar yet to be parsed
- a *parsing table* which tells it what (if any) grammar rule to apply given the symbols on top of its stack and the next input token.

The parser applies the rule found in the table by matching the top-most symbol on the stack (row) with the current symbol in the input stream (column).

When the parser starts, the stack already contains two symbols:

```
[ S, $ ]
```

where '$' is a special terminal to indicate the bottom of the stack and the end of the input stream, and 'S' is the start symbol of the grammar. The parser will attempt to rewrite the contents of this stack to what it sees on the input stream. However, it only keeps on the stack what still needs to be rewritten.

Concrete example

Set up

To explain its workings we will consider the following small grammar:

1. S → F
2. S → **(** S + F **)**
3. F → **a**

and parse the following input:

(a + a)

The parsing table for this grammar looks as follows:

	(	**)**	**a**	**+**	**$**

S	2	-	1	-	-
F	-	-	3	-	-

(Note that there is also a column for the special terminal, represented here as **$**, that is used to indicate the end of the input stream.)

Parsing procedure

In each step, the parser reads the next-available symbol from the input stream, and the top-most symbol from the stack. If the input symbol and the stack-top symbol match, the parser discards them both, leaving only the unmatched symbols in the input stream and on the stack.

Thus, in its first step, the parser reads the input symbol '**(**' and the stack-top symbol 'S'. The parsing table instruction comes from the column headed by the input symbol '**(**' and the row headed by the stack-top symbol 'S'; this cell contains '2', which instructs the parser to apply rule (2). The parser has to rewrite 'S' to '**(** S + F **)**' on the stack and write the rule number 2 to the output. The stack then becomes:

[**(**, S, +, F, **)**, **$**]

Since the '**(**' from the input stream did not match the top-most symbol, 'S', from the stack, it was not removed, and remains the next-available input symbol for the following step.

In the second step, the parser removes the '**(**' from its input stream and from its stack, since they match. The stack now becomes:

[S, +, F, **)**, **$**]

Now the parser has an '**a'** on its input stream and an 'S' as its stack top. The parsing table instructs it to apply rule (1) from the grammar and write the rule number 1 to the output stream. The stack becomes:

[F, +, F, **)**, **$**]

The parser now has an '**a'** on its input stream and an 'F' as its stack top. The parsing table instructs it to apply rule (3) from the grammar and write the rule number 3 to the output stream. The stack becomes:

[**a**, +, F, **)**, **$**]

In the next two steps the parser reads the '**a'** and '**+'** from the input stream and, since they match the next two items on the stack, also removes them from the stack. This results in:

[F, **)**, **$**]

In the next three steps the parser will replace '**F'** on the stack by '**a'**, write the rule number 3 to the output stream and remove the '**a'** and '**)'** from both the stack and the input stream. The parser thus ends with '**$'** on both its stack and its input stream.

In this case the parser will report that it has accepted the input string and write the following list of rule numbers to the output stream:

[2, 1, 3, 3]

This is indeed a list of rules for a leftmost derivation of the input string, which is:

S → **(** S + F **)** → **(** F + F **)** → **(** **a** + F **)** → **(** **a** + **a** **)**

```
/* Program to Implement LL1 Parsing */
#include<stdio.h>
#include<string.h>
char s[30],stack[20];
int main()
{
char m[5][6][3]={{"tb"," "," ","tb"," "," "},
                 {" ","+tb"," "," ","n","n"},
                 {"fc"," "," ","fc"," "," ",},
                 {" ","n","*fc"," ","n","n"},
                 {"d"," "," ","(e)"," "," "}};
int size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0,0,3,0,0};
int i,j,k,n,str1,str2;
printf("\n enter the input string:");
scanf("%s",s);
strcat(s,"$");
n=strlen(s);
stack[0]='$';
stack[1]='e';
i=1;
j=0;
printf("\n stack input\n");
printf("\n");
while((stack[i]!='$')&&(s[j]!='$'))
{
if(stack[i]=s[j])
{
i--;
j++;
}
switch(stack[i])
{
case 'e':str1=0;
        break;
case 'b':str1=1;
        break;
case 't':str1=2;
        break;
case 'c':str1=3;
```

```
        break;
case 'f':str1=4;
        break;
}
switch(s[j])
{
case 'd':str2=0;
         break;
case '+':str2=1;
         break;
case '*':str2=2;
         break;
case '(':str2=3;
         break;
case ')':str2=4;
         break;
case '$':str2=5;
         break;
}
if(m[str1][str2][0]=='$')
{
printf("\n error\n");
return(0);
}
else if(m[str1][str2][0]='n')
i--;
else if(m[str1][str2][0]='i')
stack[i]='d';
else
{
for(k=size[str1][str2]-1;k>=0;k--)
{
stack[i]=m[str1][str2][k];
i++;
}
i--;
}
for(k=0;k<=i;k++)
printf("%c",stack[k]);
printf(" ");
```

```
for(k=j;k<=n;k++)
printf("%c",s[k]);
printf("\n");
}
printf("\n SUCCESS");
}
```

OUTPUT:

```
$ cc ll1parsing.c
$ ./a.out

 enter the input string:i+i*i

 stack input

 +i*i$
 i*i$
 *i$
 i$
 $

 SUCCESS
```

6. Bottom up parsers.

Implementation Of SLR Parsing.

Concepts :

LR(0) Isn't Good Enough

LR(0) is the simplest technique in the LR family. Although that makes it the easiest to learn, these parsers are too weak to be of practical use for anything but a very limited set of grammars. The fundamental limitation of LR(0) is the zero, meaning no lookahead tokens are used. It is a stifling constraint to have to make decisions using only what has already been read, without even glancing at what comes next in the input. If we could peek at the next token and use that as part of the decision-making, we will find that it allows for a much larger class of grammars to be parsed.

SLR(1)

where the S stands for simple SLR(1) parsers use the same LR(0) configurating sets and have the same table structure and parser operation, The difference comes in assigning table actions, where we are going to use one token of lookahead to help arbitrate among the conflicts. If we think back to the kind of conflicts we encountered in LR(0) parsing, it was the reduce actions that cause us grief. A state in an LR(0) parser can have at most one reduce action and cannot have both shift and reduce instructions. Since a reduce is indicated for any completed item, this dictates that each completed item must be in a state by itself. But let's revisit the assumption that if the item is complete, the parser must choose to reduce. Is that always appropriate? If we peeked at the next upcoming token, it may tell us something that invalidates that reduction. If the sequence on top of the stack could be reduced to the non-terminal A, what tokens do we expect to find as the next input? What tokens would tell us that the reduction is not appropriate?

Perhaps Follow(A) could be useful here!

The simple improvement that SLR(1) makes on the basic LR(0) parser is to reduce only if the next input token is a member of the follow set of the non-terminal being reduced.
When filling in the table, we don't assume a reduce on all inputs as we did in LR(0), we selectively choose the reduction only when the next input symbols in a member of the follow set. To be more precise, here is the algorithm for SLR(1) table construction (note all steps are the same as for LR(0) table construction except for 2a)

1. Construct F = {I0, I1, ... In}, the collection of LR(0) configurating sets for G'.
2. State i is determined from Ii
. The parsing actions for the state are determined as follows:
2 a) If A –> u• is in Ii
then set Action[i,a] to reduce A –> u for all a in Follow(A) (A is not
S').
b) If S' –> S• is in Ii
then set Action[i,$] to accept.
c) If A –> u•av is in Ii
and successor(Ii
, a) = Ij
, then set Action[i,a] to shift j (a must be
a terminal).
3. The goto transitions for state i are constructed for all non-terminals A using the
rule: If successor(Ii,
A) = Ij,
then Goto [i, A] = j.
4. All entries not defined by rules 2 and 3 are errors.
5. The initial state is the one constructed from the configurating set containing S' –>
•S.

In the SLR(1) parser, it is allowable for there to be both shift and reduce items in the same state as well as multiple reduce items. The SLR(1) parser will be able to determine
which action to take as long as the follow sets are disjoint.

Let's consider those changes at the end of the LR(0) handout to the simplified expression grammar that would have made it no longer LR(0). Here is the version with the addition
of array access:

E' –> E
E –> E + T | T
T –> (E) | id | id[E]

Here are the first two LR(0) configurating sets entered if id is the first token of the input.
In an LR(0) parser, the set on the right has a shift-reduce conflict. However, an SLR(1)
will compute Follow(T) = { +)] $ } and only enter the reduce action on those tokens. The
input [will shift and there is no conflict. Thus this grammar is SLR(1) even though it is
not LR(0).

Similarly, the simplified expression grammar with the assignment addition:

E' –> E
E –> E + T | T | V = E
T –> (E) | id
V –> id

E' -> •E
E -> •E + T
E -> •T
T -> •(E)
T -> •id
T -> •id[E]

T -> id•
T -> id•[E]
id 3
Here are the first two LR(0) configurating sets entered if id is the first token of the input.

In an LR(0) parser, the set on the right has a reduce-reduce conflict. However, an SLR(1)
parser will compute Follow(T) = { +) $ } and Follow(V) = { = } and thus can distinguish which
reduction to apply depending on the next input token. The modified grammar is SLR(1).
SLR(1) Grammars
A grammar is SLR(1) if the following two conditions hold for each configurating set:
1. For any item A –> u•xv in the set, with terminal x, there is no complete item B –> w•
in that set with x in Follow(B). In the tables, this translates no shift-reduce conflict on
any state. This means the successor function for x from that set either shifts to a
new state or reduces, but not both.
2. For any two complete items A –> u• and B –> v• in the set, the follow sets must be
disjoint, e.g. Follow(A) ∩ Follow(B) is empty. This translates to no reduce-reduce
conflict on any state. If more than one non-terminal could be reduced from this set,
it must be possible to uniquely determine which using only one token of
lookahead.

All LR(0) grammars are SLR(1) but the reverse is not true, as the two extensions to our

expression grammar demonstrated. The addition of just one token of lookahead and use
of the follow set greatly expands the class of grammars that can be parsed without conflict.

```
/* Program to Implement SLR Parsing */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int axn[][6][2]={
 {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
 {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
 {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
 {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
 {{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
 {{100,5},{-1 -1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
 {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
 {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
 {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
 {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}},
};

int gotot[12][3]={1,2,3,-1,-1,-1-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,-1,9,3,-1,-1,10,
                  -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
int a[10];
char b[10];
int top=-1,btop=-1,i;
void push(int k)
{
if(top<9)
a[++top]=k;
}

void pushb(char k)
{
if(btop<9)
b[++btop]=k;
}

char TOS()
{
return a[top];
}
```

```
void pop()
{
if(top>=0)
top--;
}

void popb()
{
if(btop>=10)
b[btop--]='\0';
}
void display()
{
for(i=0;i<top;i++)
printf("%d%c",a[i],b[i]);
}
void display1(char p[],int m)
{
int l;
printf("\t\t");
for(l=m;p[l]!='\0';l++)
printf("%c",p[l]);
printf("\n");
}
void error()
{
printf("syntax error");
}
void reduce(int p)
{
int k,ad;
char src,*dest;
switch(p)
{
case 1:dest="E+T";
       src='E';
       break;
case 2:dest="T";
       src='E';
```

```
        break;

case 3:dest="T*F";
        src='T';
        break;
case 4:dest="F";
        src='T';
        break;
case 5:dest="(E)";
        src='F';
        break;
case 6:dest="i";
        src='F';
        break;
default :dest="\0";
         src='\0';
         break;
}
for(k=0;k<strlen(dest);k++)
{
pop();
popb();
}
pushb(src);
switch(src)
{
case 'E':ad=0;
            break;
case 'T':ad=1;
            break;
case 'F':ad=2;
            break;
default:ad=-1;
            break;
}
push(gotot[TOS()][ad]);
}

int main()
{
```

```
int j,st,ic;
char ip[20]="\0",an;
printf("enter any string");
scanf("%s",ip);
push(0);
display();
printf("\t%s\n",ip);
for(j=0;ip[j]!='\0';)
{
st=TOS();
an=ip[j];
if(an>='a'&&an<='z') ic=0;
else if(an=='+') ic=1;
else if(an=='*') ic=2;
else if(an=='(') ic=3;
else if(an==')') ic=4;
else if(an=='$') ic=5;
else
{
error();
break;
}

if(axn[st][ic][0]==100)
{
pushb(an);
push(axn[st][ic][1]);
display();
j++;
display1(ip,j);
}
if(axn[st][ic][0]==101)
{
reduce(axn[st][ic][1]);
display();
display1(ip,j);
}
if(axn[st][ic][1]==102)
{
printf("given string is accepted");
```

```
break;
}
}
return(0);
}
```

OUTPUT:

```
$ cc slr.c
$ ./a.out
enter any stringi+i*i
      i+i*i
0i           +i*i
0i5          +i*i
0i53          +i*i
```

7. Parser Generation using YACC.

YACC Concepts:

The unix utility yacc (Yet Another Compiler Compiler) parses a stream of token, typically generated by lex, according to a user-specified grammar.
2 Structure of a yacc file
A yacc file looks much like a lex file:
...definitions...
%%
...rules...
%%
...code...
definitions : All code between %{ and %} is copied to the beginning of the resulting C file.
Rules : A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.
code : This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyser. If the code segment is left out, a default main is used which only calls yylex.
3 Definitions section
There are three things that can go in the definitions section:
C code Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.
definitions: The definitions section of a lex file was concerned with characters; in yacc this is tokens. These token definitions are written to a .h file when yacc compiles this file.
associativity rules : These handle associativity and priority of operators.

Lex Yacc interaction

Conceptually, lex parses a file of characters and outputs a stream of tokens; yacc accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled. If your lex program is supplying a tokenizer, the yacc program will repeatedly call the yylex routine. The lex rules will probably function by calling return everytime they have parsed a token.

If lex is to return tokens that yacc will process, they have to agree on what tokens there are.

This is done as follows.

• The yacc file will have token definitions

%token NUMBER

in the definitions section.

• When the yacc file is translated with yacc -d, a header file y.tab.h is created

that has definitions like

#define NUMBER 258

This file can then be included in both the lex and yacc program.

• The lex file can then call return NUMBER, and the yacc program can match on this token.

The return codes that are defined from %TOKEN definitions typically start at around 258, so that single characters can simply be returned as their integer value:

```
/* in the lex program */
[0-9]+ {return NUMBER}
[-+*/] {return *yytext}
/* in the yacc program */
sum : TERMS '+' TERM
```

Return values:

In addition to specifying the return code, the lex parse can return a symbol that is put on top of the stack, so that yacc can access it. This symbol is returned in the variable yylval. By default, this is defined as an int, so the lex program would have extern int llval;

```
%%
[0-9]+ {llval=atoi(yytext); return NUMBER;}
```

If more than just integers need to be returned, the specifications in the yacc code become more complicated. Suppose we want to return double values, and integer indices in a table.
The following three actions are needed.
1. The possible return values need to be stated:
%union {int ival; double dval;}
2. These types need to be connected to the possible return tokens:
2%token <ival> INDEX
%token <dval> NUMBER
3. The types of non-terminals need to be given:
%type <dval> expr
%type <dval> mulex
%type <dval> term
The generated .h file will now have
#define INDEX 258
#define NUMBER 259
typedef union {int ival; double dval;} YYSTYPE;
extern YYSTYPE yylval;

Rules section:
The rules section contains the grammar of the language you want to parse. This looks like
name1 : THING something OTHERTHING {action}
| othersomething THING {other action}
name2 : .....
This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.
The terminal symbols get matched with return codes from the lex tokenizer. They are typically defines coming from %token definitions in the yacc program or character values;

User code section:
The minimal main program is

```
int main()
{
yyparse();
return 0;
}
```

Extensions to more ambitious programs should be self-evident.
In addition to the main program, the code section will usually also contain subroutines, to be used either in the yacc or the lex program.

```
/* YACC Program of an advanced desk calculator */
%{
#include<ctype.h>
#include<stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left'+"-'
%left'*"/'
%right UMINUS
%%
lines:lines expr'\n'
{
printf("%g\n",$2);}
|lines'\n'
|/*E*/
;
expr:expr'+'expr{$$=$1+$3;}
|expr'-'expr{$$=$1-$3;}
|expr'*'expr{$$=$1*$3;}
|expr'/'expr{$$=$1/$3;}
|'('expr')'{$$=$2;}
|'-'expr%prec UMINUS{$$=-$2;}
|NUMBER
;
%%
```

```
yylex()
{
int c;
while((c=getchar())==' ');
if((c=='.')||(isdigit(c))){
ungetc(c,stdin);
scanf("%lf",&yylval);
return NUMBER;
}
return c;
}
int main()
{
yyparse();
return 1;
}
int yyerror()
{
return 1;
}
int yywrap()
{
return 1;
}
```

OUTPUT:

```
$ yacc calc.y
$ gcc -o calc y.tab.c
$ ./calc

1+2/3*6
5
```

8. Intermediate Code Generation

Program For 3 Address Code.
Concepts :

In computer science, **three-address code** (often abbreviated to TAC or 3AC) is a form of representing intermediate code used by compilers to aid in the implementation of code-improving transformations. Each instruction in three-address code can be described as a 4-tuple: (operator, operand1, operand2, result).

Each statement has the general form of:

$\text{result} := \text{operand}_1 \ \text{operator} \ \text{operand}_2$

such as:

$x := y \ \text{op} \ z$

where *x*, *y* and *z* are variables, constants or temporary variables generated by the compiler. *op* represents any operator, *e.g.* an arithmetic operator.

Expressions containing more than one fundamental operation, such as:

$p := x + y \times z$

are not representable in three-address code as a single instruction. Instead, they are decomposed into an equivalent series of instructions, such as

$t_1 := y \times z$

$p := x + t_1$

The term *three-address code* is still used even if some instructions use more or fewer than two operands. The key features of three-address code are that every instruction implements exactly one fundamental operation, and that the source and destination may refer to any available register.

A refinement of three-address code is static single assignment form (SSA).

/* Program to Implement 3 Address Code */

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l;
char ex[10],ex1[10],exp1[10],ex2[10];
main()
{
while(1)
{
printf("\n 1.Assignment\n 2.Arithmatic\n 3.exit\n ENTER THE
CHOICE:");
scanf("%d",&ch);
switch(ch)
{
case 1:printf("\n enter the expression with assignment operator:");
        scanf("%s",ex1);
        l=strlen(ex1);
        ex2[0]='\0';
        i=0;
        while(ex1[i]!='=')
        {
        i++;
        }
        strncat(ex2,ex1,i);
        strrev(ex1);
        exp1[0]='\0';
        strncat(exp1,ex1,l-(i+1));
        strrev(exp1);
        printf("3 address code:\n temp=%s \n %s=temp\n",exp1,ex2);
        break;
```

```
case 2:printf("\n enter the expression with arithmatic operator:");
        scanf("%s",ex);
        strcpy(ex1,ex);
        l=strlen(ex1);
        exp1[0]='\0';
        for(i=0;i<l;i++)
        {
        if(ex1[i]=='+'||ex1[i]=='-')
        {
        if(ex1[i+2]=='/'||ex1[i+2]=='*')
        {
        pm();
        break;
        }
        else
        {
        plus();
        break;
        }
        }
        else if(ex1[i]=='/'||ex1[i]=='*')
        {
        div();
        break;
        }
        }
        break;
        }
        }
        break;
case 3:exit(0);
        }
        }
        }
void pm()
```

```
{
strrev(exp1);
j=l-i-1;
strncat(exp1,ex1,j);
strrev(exp1);
printf("3 address code:\n temp=%s\n temp1=%c%c
temp\n",exp1,ex1[j+2],ex1[j]);
}

void div()
{
strncat(exp1,ex1,i+2);
printf("3 address code:\n temp=%s\n
temp1=temp%c%c\n",exp1,ex1[l+2],ex1[i+3]);
}
void plus()
{
strncat(exp1,ex1,i+2);
printf("3 address code:\n temp=%s\n
temp1=temp%c%c\n",exp1,ex1[l+2],ex1[i+3]);
}
```

OUTPUT:

```
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:1
Enter the exp with assignment operator:a=b
3 address code:
temp=b
a=temp
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:2
Enter the exp with arithmetic operator:a*b+c
3 address code:
temp=a*b
temp1=temp+c
1.Assignment
2.Arithmetic
3.Exit
Enter the choice:3
```

9. Target Code Generation.

In computer science, **code generation** is the process by which a compiler's **code generator** converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the *backend*) needs to change from target to target. (For more information on compiler design, see Compiler.)

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code. Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

- Instruction selection: which instructions to use.

- Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.
- Register allocation: the allocation of variables to processor registers[1]
- Debug data generation if required so the code can be debugged.

Instruction selection is typically carried out by doing a recursive postorder traversal on the abstract syntax tree, matching particular tree configurations against templates; for example, the treeW := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

In a compiler that uses an intermediate language, there may be two instruction selection stages — one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding opcodes. However, if the compiler is actually a language translator (for example, one that converts Eiffel to C), then the second code-generation phase may involve *building* a tree from the linear intermediate code.

```
/* Program to Implement Code Generation */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int label[20];
int no=0;
int main()
{
FILE *fp1,*fp2;
char fname[10],op[10],ch;
char operand1[8],operand2[8],result[8];
int i=0,j=0;
printf("\n enter the filename of intermediate code\n");
scanf("%s",&fname);
fp1=fopen(fname,"r");
fp2=fopen("target.txt","w");
if(fp1==NULL||fp2==NULL)
{
printf("\n error opening the file");
exit(0);
}
while(!feof(fp1))
{
fprintf(fp2,"\n");
fscanf(fp1,"%s",op);
i++;
if(check_label(i))
fprintf(fp2,"\nlabel#%d",i);
if(strcmp(op,"print")==0)
{
fscanf(fp1,"%s",result);
fprintf(fp2,"\n\tOUT%s",result);
}
if(strcmp(op,"goto")==0)
{
fscanf(fp1,"%s%s",operand1,operand2);
label[no++]=atoi(operand2);
}
if(strcmp(op,"[]=")==0)
{
```

```
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\tSTORE%s[%s],%s",operand1,operand2,result);
}
if(strcmp(op,"uminus")==0)
{
fscanf(fp1,"%s,%s",operand1,result);
fprintf(fp2,"\n\t,LOAD%s",operand1);
fprintf(fp2,"\n\tSTORE R1,%s",result);
}
switch(op[0])
{
case '*':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t MUL R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '+':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD%s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t ADD R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '-':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD%s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t SUB R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '/':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD%s,R0",operand1);
fprintf(fp2,"\n\t LOAD %s,R1",operand2);
fprintf(fp2,"\n\t DIV R1,R0");
fprintf(fp2,"\n\t STORE R0,%s",result);
break;
case '=':
```

```
fscanf(fp1,"%s%s",operand1,result);
fprintf(fp2,"\n\t STORE %s%s",operand1,result);
break;
case '>':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD%s,R0",operand1);
fprintf(fp2,"\n\t JGT%S,label#%s",operand2,result);
label[no++]=atoi(result);
break;
case '<':
fscanf(fp1,"%s%s%s",operand1,operand2,result);
fprintf(fp2,"\n\t LOAD%s,R0",operand1);
fprintf(fp2,"\n\t JLT%S,label#%s",operand2,result);
label[no++]=atoi(result);
break;
}
}
fclose(fp2);
fclose(fp1);
fp2=fopen("target.txt","r");
if(fp2==NULL)
{
printf("error opening in file\n");
exit(0);
}
do{
ch=fgetc(fp2);
printf("%c",ch);
}
while(ch!=EOF);
fclose(fp1);
return 0;
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
```

```
}
return 0;
}
```

Input.txt

```
/t3 t2 t2
uminus t2 t2
print t2
+t1 t3 t4
print t4
```

OUTPUT:

```
$ cc codegen.c
$ ./a.out

 enter the filename of intermediate code
input.txt

        LOADt2,R0
        LOAD t2,R1
        DIV R1,R0
        STORE R0,uminus


       OUTt2

        LOADt3,R0
        LOAD t4,R1
        ADD R1,R0
        STORE R0,print
```

$vi target.txt

```
LOADt2,R0
LOAD t2,R1
DIV R1,R0
STORE R0,uminus

OUTt2

LOADt3,R0
LOAD t4,R1
ADD R1,R0
STORE R0,print
```

10. Code optimization.

Program For Code Optimization Using Constant Folding.

Concepts:
Constant Folding

Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time.

Example:

In the code fragment below, the expression (3 + 5) can be evaluated at compile time and replaced with the constant 8.

```
int f (void)
{
  return 3 + 5;
}
```

Below is the code fragment after constant folding.

```
int f (void)
{
  return 8;
}
```

Notes:

Constant folding is a relatively easy optimization.

Programmers generally do not write expressions such as (3 + 5) directly, but these expressions are relatively common after macro expansion and other optimizations such as constant propagation.

All C compilers can fold integer constant expressions that are present after macro expansion (ANSI C requirement). Most C compilers can fold integer constant expressions that are introduced after other optimizations.

Some environments support several floating-point rounding modes that can be changed dynamically at run time. In these

environments, expressions such as (1.0 / 3.0) must be evaluated at run-time if the rounding mode is not known at compile time

/* Program to Implement Code Optimization using Constant Folding */

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
struct ConstFold {
char new_str[10];
char str[10];
} Opt_Data[20];

void ReadInput(char Buffer[], FILE *Out_file);
int Gen_token(char str[], char Tokens[][10]);

int New_Index = 0;

int main() {
FILE *In_file, *Out_file;
char Buffer[100], ch;
int i = 0;
In_file = fopen("code.txt", "r");
Out_file = fopen("output.txt", "w");
while(1) {
ch = fgetc(In_file);
i = 0;
while(1) {
if(ch == '\n') break;
Buffer[i++] = ch;
ch = fgetc(In_file);
if(ch == EOF) break;
}
if(ch == EOF) break;
Buffer[i] = '\0';
ReadInput(Buffer, Out_file);
}
```

```
return 0;
}

void ReadInput(char Buffer[], FILE *Out_file) {
char temp[100], Token[10][10];
int n, i, j, flag = 0;
strcpy(temp, Buffer);
n = Gen_token(temp, Token);
for(i=0; i<n; i++) {
if(!strcmp(Token[i], "=")) {
if(isdigit(Token[i+1][0]) || Token[i+1][0] == '.')
{
flag = 1;
strcpy(Opt_Data[New_Index].new_str, Token[i-1]);
strcpy(Opt_Data[New_Index++].str, Token[i+1]);
}
}
}
if(!flag) {
for(i=0; i<New_Index; i++) {
for(j=0; j<n; j++) {
if(!strcmp(Opt_Data[i].new_str, Token[j]))
strcpy(Token[j], Opt_Data[i].str);
}
}
}

fflush(Out_file);
strcpy(temp, "");
for(i=0; i<n; i++) {
strcat(temp, Token[i]);
if(Token[i+1][0]!=',' || Token[i+1][0]!=';')
strcat(temp, "");
}
strcat(temp, "\n\0");
fwrite(&temp, strlen(temp), 1, Out_file);
}

int Gen_token(char str[],char Token[][10])
{
```

```
int i=0, j=0, k=0;
while(str[k]!= '\0') {
j=0;
while(str[k]==' ' || str[k] == '\t')
k++;
while(str[k]!= ' '&&str[k]!='\0'&&str[k]!='='&&str[k]!= '/'&&str[k]!='+
'&&str[k]!='-'&&str[k]!='*'&&str[k]!=','&&str[k]!= ';')

Token[i][j++] = str[k++];
Token[i++][j] = '\0';
if(str[k] == '=' || str[k] == '/' || str[k] == '+' || str[k] == '-' ||
str[k] == '*' || str[k] == ',' || str[k] == ';')
{
Token[i][0] = str[k++];
Token[i++][1] = '\0';
}
if(str[k] == '\0')
break;
}
return i;
}
```

Input.txt

```
#include<stdio.h>
main()
{
float pi=3.14,r,a;
a=pi*r*r;
printf("a=%f",a);
return 0;
}
```

OUTPUT:

```
$ cc codeop.c
$ ./a.out
$ vi output.txt
```

Output.txt

```
#include<stdio.h>
main()
{
floatpi=3.14,r,a;
a=3.14*r*r;
printf("a=%f",a);
return0;
}
```