Ex.No.6 COMPUTE FIRST AND FOLLOW FUNCTION USING PREDICTIVE PARSING

DATE:

AIM

To find first and follow of a given context free grammar

THEORY:

Why FIRST?

To avoid backtracking in parsing we need to calculate first.

S -> cAd

A -> bc|a

And the input string is "cad".

If the compiler would have come to know in advance, that what is the "first character of the string produced when a production rule is applied", and comparing it to the current character or token in the input string it sees, it can wisely take decision on which production rule to apply.

Computing the Function *FIRST*

If X is Grammar Symbol, then First (X) will be –

- If X is a terminal symbol, then FIRST(X) = {X}
- If $X \rightarrow \varepsilon$, then FIRST(X) = {ε}
- If X is non-terminal & $X \rightarrow a\ \alpha$, then FIRST (X) = {a}
- If $X \rightarrow Y_1, Y_2, Y_3$, then FIRST (X) will be

(a) If Y is terminal, then

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = {Y_1}

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then,

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1)

(c) If FIRST (Y_1) contains ε, then.

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) – {ε} ∪ FIRST(Y_2, Y_3)

Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) – {ε} ∪ FIRST (Y_3)

Why FOLLOW?

The parser faces one more problem. Let us consider below grammar to understand this problem.

A -> aBb

B -> c | ε

And suppose the input string is "ab" to parse.

As the first character in the input is a, the parser applies the rule A->aBb.

```
   A
  /|\
 a B b
```

Now the parser checks for the second character of the input string which is b, and the Non-Terminal to derive is B, but the parser can't get any string derivable from B that contains b as first character.

But the Grammar does contain a production rule B -> ε, if that is applied then B will vanish, and the parser gets the input "ab", as shown below. But the parser can apply it only when it knows that the character that follows B in the production rule is same as the current character in the input.

In RHS of A -> aBb, b follows Non-Terminal B, i.e. FOLLOW(B) = {b}, and the current input character read is also b. Hence the parser applies this rule. And it is able to get the string "ab" from the given grammar.

```
   A            A
  /|\          / \
 a B b  =>    a   b
   |
   ε
```

So FOLLOW can make a Non-terminal vanish out if needed to generate the string from the parse tree.

Computing the Function *FOLLOW*

1. First, put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. Suppose there is a production rule of A → aBB, (where a can be a whole string) then everything in FIRST(B) except for ε is placed in FOLLOW(B).
3. Suppose there is a production rule of A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
4. Suppose there is a production rule of A → aBC, where FIRST(C) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

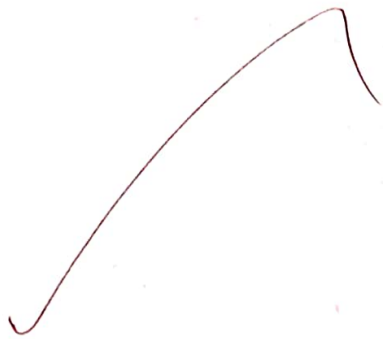
Program:

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
int n, m = 0, i = 0, j = 0;
char a[10][10], f[10];
void follow (char c);
void first (char c);
int main(){
    int z;
    char c, ch;
    printf("Enter the number of productions: \n");
    scanf("%d", &n);
    printf("Enter the productions : \n");
    for(i=0; i<n; i++){
        scanf("%s", a[i]);
    }
    do {
        m=0;
        printf("Enter the element whose FIRST & FOLLOW
                is to be found: ");
        scanf("%c", &c);
        first(c);
        printf("First(%c) ={", c);
        for(i=0; i<m; i++)
            printf("%c", f[i]);
```

```
        printf("}\n");
        f[0] = '\0';
        m=0;
        follow(c);
        printf("Follow(%c) = {", c);
        for(i=0; i<m; i++){
            printf("%c", f[i]);
        }
        printf("}\n");
        printf("continue (0/1)? ");
        scanf("%d", &z);
    }while(z==1);
    return 0;
}
void first(char c){
    int k;
    if(!isupper(c)){
        f[m++] = c;
        return;
    }
    for(k=0; k<n; k++){
        if(a[k][0]==c){
            if(a[k][2] == '$'){
                follow(a[k][0]);
            }else if(islower(a[k][2])){
                f[m++] = a[k][2];
            }else{
                first(a[k][2]);
```

```
    }
}
}
}
```

```
void follow (char c){
    if (a[0][0] == c){
        f[m++] = '$';
    }
    for (i=0; i<n; i++){
        for (j=2; j< strlen(a[i]); j++){
            if (a[i][j] == c){
                if (a[i][j+1] != '\0'){
                    first (a[i][j+1]);
                }
                if (a[i][j+1] == '\0' && c != a[i][0])
                {
                    follow (a[i][0]);
                }
            }
        }
    }
}
```



SAMPLE INPUT

Consider the expression grammar ,

A -> aAaB

B -> bBaB

A-> a

B -> b

SAMPLE OUTPUT

First (A) = a
First (B) = b
Follow (A)= $, a, b
Follow (B) =$, a, b

VIVA QUESTIONS

1. What is the need of calculating FIRST?
2. What is the need for FOLLOW?
3. Compare Top Down and Bottom-up parser?
4. Does Top-Down Parser handle Left Recursive Grammar?
5. State the rule for eliminating Left Recursion.

RESULT:

Thus the implementation of FIRST and FOLLOW Function using predictive parsing has been Successfully executed and the Output was verified.

(10) Good

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) (Technical – 5 and Communications - 5)	
Total (100)	

Ex.No.: 7 **CONSTRUCTION OF PREDICTIVE PARSING TABLE**

DATE:

AIM

To implement the Predictive Parsing Table using any programing language.

ALGORITHM:

1. Read Grammar rules and store them
2. Create arrays for First, Follow and parsing table
3. Find First for each non-terminal.
4. Find Follow using First and grammar rules.
5. Initialize table with non-terminals and terminals.
6. place productions in table based on First set.
7. use Follow set if ϵ (epsilon) is in First.
8. Fill empty cells with -.
9. print the predictive parsing table.
10. Apply table for LL(1) parsing.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 10
#define SIZE 10

typedef struct {
    char nonTerminal;
    char production[SIZE][SIZE];
    int prodCount;
} Grammar;

Grammar grammar[MAX];
char first[MAX][SIZE], follow[MAX][SIZE]
char parsingTable[MAX][SIZE][SIZE];
int grammarCount = 0;

void findFirst(char symbol, char result[]){
    if(!(symbol >= 'A' && symbol <= 'Z')){
        strncat(result, &symbol, 1);
        return;
    }
    for(int i=0; i< grammarCount; i++){
        if(grammar[i].nonTerminal == symbol){
            for(int j=0; j< grammar[i].prodCount; j++){
                findFirst(grammar[i].production[j][0], result);
            }
        }
    }
}
```

```
Void findFollow (char Symbol, char result[]){
    if (symbol == grammar[0].nonTerminal){
        strcat (result, "$");
    }
    for(int i=0; i< grammarCount; i++){
        for (int j=0; j< grammar[i].prodCount; j++){
            char *pos = strchr (grammar[i].production[j].
                                                Symbol);
            if (pos != NULL){
                char next = *(pos+1);
                if (next != '\0'){
                    findFirst (next, result);
                } else if (grammar[i].nonTerminal != Symbol)
                {
                    findFollow (grammar[i].nonTerminal, result);
                }
            }
        }
    }
}
Void constructParsingTable(){
    for (int i=0; i< grammarCount; i++){
        for (int j=0; j< grammar[i].prodCount; j++){
            char firstSet [SIZE] = "";
            findFirst (grammar[i].production[j][0], firstSet);
            for (int k=0; k< strlen (firstSet); k++){
                if (firstSet[k] != 'e'){
                    strcpy (parsingTable[i][firstSet[k] - 'a'],
                                grammar[i].production[j]);
                }
```

```
        else{
            char followSet[SIZE] = "";
            findFollow(grammar[i].nonTerminal, followSet);
            for(int l=0; l< strlen(followSet); l++){
                strcpy(parsingTable[i][followSet[l]-'a'], "e");
            }
        }
    }
}
}

Void displayParsingTable(){
    printf("\n predictive parsing Table:\n");
    printf("--------------------\n");
    printf("|NT| id | + | * | ( | ) | $ |\n");
    printf("--------------------\n");
    for(int i=0; i< grammarCount; i++){
        printf("| %c ", grammar[i].nonTerminal);
        char terminals[] = {'i', '+', '*', '(', ')', '$'};
        for(int j=0; j<6; j++){
            int index = terminals[j] - 'a';
            if(parsingTable[i][index][0] != '\0'){
                printf("| %s ", parsingTable[i][index]);
            }else{
                printf("|- ");
            }
        }
        printf("|\n");
    }
```

```
    printf("----------\n");
}
int main(){
  printf("Enter number of grammar rules:");
  scanf("%d", &grammarCount);
   getchar();
  for(int i=0; i<grammarCount; i++){
      printf("Enter non-Terminal:");
      scanf("%d", &grammar[i].prodCount);
      getchar();

     for (int j=0; j<grammar[i].prodCount; j++){
        printf("production %d:", j+1);
        scanf("%[^\"]s", grammar[i].production[j]);
     }
  }
  constructParsingTable();
  displayParsingTable();
  return 0;
  }
```

Output :-

```
Enter number of grammar rules: 5
Enter non-terminal : E
Enter number of production for E : 1
production 1 : TX
Enter non-terminal : X
Enter number of productions for X : 2
production 1 : +TX
production 2 : e
Enter non-Terminal : T
Enter number of productions for T : 1
production 1 : FY
Enter non-terminal : F
Enter number of productions for F : 2
production 1 : (E)
production 2 : id
```

Output :

```
Enter non-terminal : Y
Enter number of productions for Y: 2
production1 : *FY
production2 : e
predictive parsing table.
|NT| id | + | * | ( | ) | $ |
|E | TX | - | - |TX | - | - |
|X | - | +TX | - | - | e | e |
|T | FY | - | - |FY | - | - |
|F | id | - | - |(E)| - | - |
|Y | - | e |*FY| - | - | - |
```

VIVA QUESTIONS

1. Does Predictive Parser can handle Left Recursive Grammar?
2. What do you mean Left Recursive Grammar?
3. Does Predictive Parser can handle Left Factoring Grammar?
4. What do you mean Left Factoring Grammar?
5. What do you mean LL [1] Grammar?

RESULT:

The implementation of predictive parsing table using C programming Language has been successfully completed and the output was verified.

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) **(Technical – 5 and Communications - 5)**	
Total (100)	

Ex.No.:8 CONSTRUCTION OF SLR PARSING TABLE

DATE:

AIM

To write a program for implementing SLR bottom up parser for the given grammar

THEORY:

LR parsers:

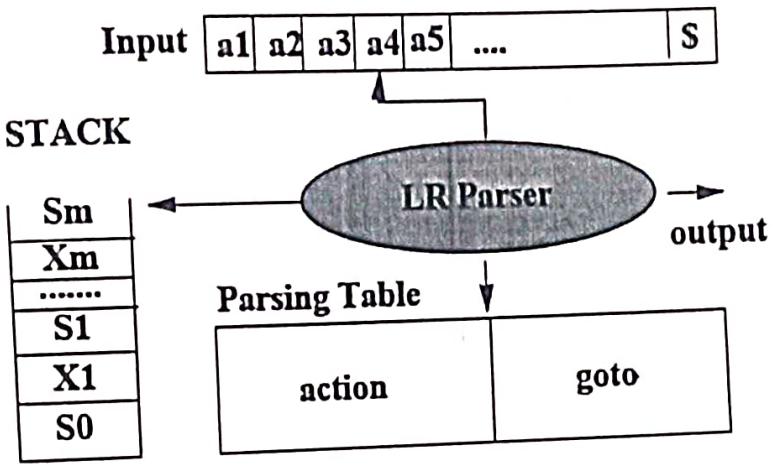


Fig: LR Parser

It is an efficient bottom-up syntax analysis technique that can be used to parse large classes of context free grammar is called LR(o) parsing.

L stands for the left to right scanning

R stands for rightmost derivation in reverse

o stands for no. of input symbols of lookahead

Advantages of LR parsing:

- It recognizes virtually all programming language constructs for which CFG can be written
- It is able to detect syntactic errors
- It is an efficient non-backtracking shift reducing parsing method.

Types of LR parsing methods:

1. SLR
2. CLR
3. LALR

STEP3 – Find FOLLOW of LHS of production

FOLLOW(S)=$

FOLLOW(A)=a,b,$

STEP 4-

Defining 2 functions: goto[list of non-terminals] and action[list of terminals] in the parsing table. Below is the SLR parsing table.

Algorithm

Input – An Augmented Grammar G'

Output – SLR Parsing Table

Method

- Initially construct set of items

$C = \{I_0, I_1, I_2 \dots \dots I_n\}$ where C is a set of LR (0) items for Grammar.

- Parsing actions are based on each item or state I_i.

Various Actions are –

- If $A \rightarrow \alpha \cdot a \beta$ is in I_i and goto $(I_i, a) = I_j$ then set Action [i, a] = shift j".
- If $A \rightarrow \alpha \cdot$ is in I_i then set Action [i, a] to "reduce $A \rightarrow \alpha$" for all symbol a, where $a \in$ FOLLOW (A).
- If $S' \rightarrow S \cdot$ is in I_i then the entry in action table Action [i, $] = accept".
- The goto part of the SLR table can be filled as– The goto transition for the state i is considered for non-terminals only. If goto $(I_i, A) = I_j$ then goto [i, A] = j
- All entries not defined by rules 2 and 3 are considered to be "error. "

	ACTION			GOTO	
	a	b	$	A	S
0	S3	S4		2	1
1			accept		
2	S3	S4		5	
3	S3	S4		6	
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX 100
#define STATES 3
#define SYMBOLS 3
```

```
int parse_table [STATES] [SYMBOLS];
void construct_parse_table(){
    for(int i=0; i<STATES; i++){
        for (int j=0; j< SYMBOLS; j++){
            Parse_table [i] [j] = -1;
        }
    }
   parese_table [0][0] = 1;
   parse_table [1][1] = 2;
   parse_table [2] [2] = 3;
}
void print_parse_table(){
    printf("\n parsing Table : \n");
    printf(" a  b $\n");
    for (int i=0; i< SYMBOLS; i++){
        printf("%.d", i);
    }
    for (int j=0; j<STATES; j++){
        if (parse_table[i][j] == -1)
            printf("- ");
        if (parse_table [i][j]==1)
            printf("S1");
        if (parse_table [i][j]==2)
            printf("S2");
        if (parse_table [i][j] == 3)
            printf ("Accept");
        }
        printf("\n");
    }
}
```

```
Void print_productions(){
    printf("\n Grammar productions:\n");
    printf("S -> aB\n");
    printf("B -> b\n");
}
char stack[MAX];
int top = -1;
Void push(char c){
    stack[++top]=c;
}
char pop(){
  return (top == -1) ? '$' : stack[top--];
 }
 Void displayStack(){
     for(int i=0; i<=top; i++){
         printf("%c", stack[i]);
     }
     printf("\n");
}

int main(){

    Construct_parse_table();
    print_parse_table();
    print_productions();

  char  input[MAX];
```

```
int state=0, i=0;
printf("\n Enter the string to be parsed: ");
scanf("%s", input);
strcat(input, "$");
push('0');
while(input[i] != '\0'){
    int symbol;
    if(input[i]=='a')
        symbol=0;
    else if(input[i]=='b')
        symbol=1;
    else if(input[i]=='$')
        symbol=2;
    else{
        printf("Invalid symbol encountered!\n");
        return 1;
    }
    int action = parse_table[state][symbol];
    if(action==-1){
        printf("String not accepted!\n");
        return 1;
    }
    else{
        push(input[i]);
        state = action;
    }
    i++;
}
```

```
if (state==3){
    printf(" String accepted!\n");
}else{
    printf("string not accepted!\n");
}
return 0;
}
```

Output:-

parsing Table:

	a	b	$
0	S_1	-	-
1	-	S_2	-
2	-	-	Accept

Grammar productions:

$S \rightarrow aB$

$B \rightarrow b$

Enter the string to be parsed: ab

String Accepted!

OUTPUT:

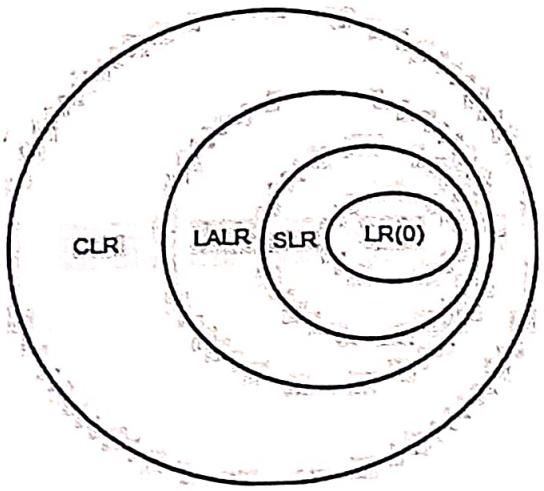
Enter any Stringv a+a*a$

	a+a*a$
0	+a*a$
0a5	+a*a$
0F3	+a*a$
0T2	+a*a$
0E1	a*a$
0E1+6	*a$
0E1+6a5	*a$
0E1+6F3	*a$
0E1+6T9	a$
0E1+6T9*7	$
0E1+6T9*7a5	$
0E1+6T9*7F10	$
0E1+6T9	$
0E1	

Given String is accept

NOTE:

1. Even though CLR parser does not have RR conflict but LALR may contain RR conflict.
2. If number of states LR(0) = n1,
   number of states SLR = n2,
   number of states LALR = n3,
   number of states CLR = n4 then,
   n1 = n2 = n3 <= n4



VIVA QUESTIONS

1. Define LR Parser.
2. State the advantages of LR Parser.
3. List the types of LR Parser.
4. Compare LR and LL Parser.
5. What do you mean by GOTO operation?

RESULT:

The above program for SLR parser is executed successfully and the output was verified.

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) (Technical – 5 and Communications - 5)	
Total (100)	

Ex.No.:9

CONSTRUCT THE THREE ADDRESS CODE FOR THE GIVEN EXPRESSION

DATE:

AIM

To write program to construct the three-address code for the given expression

THEORY:

Three address code

- Three-address code is an intermediate code. It is used by the Code Optimizer.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

They use maximum three addresses to represent any statement. They are implemented as a record with the address fields.

General Form-

In general, Three Address instructions are represented as-

a = b op c

Here,

- a, b and c are the operands.
- Operands may be constants, names, or compiler generated temporaries.
- op represents the operator.

Examples-

Examples of Three Address instructions are-

- a = b + c
- c = a x b

Common Three Address Instruction Forms-

The common forms of Three Address instructions are-

1. Assignment Statement-

x = y op z and x = op y

Here,

- x, y and z are the operands.
- op represents the operator.

It assigns the result obtained after solving the right side expression of the assignment operator to the left side operand.

2. Copy Statement-

```
x = y
```

Here,

- x and y are the operands.
- = is an assignment operator.

It copies and assigns the value of operand y to operand x.

3. Conditional Jump-

```
If x relop y goto X
```

Here,

- x & y are the operands.
- X is the tag or label of the target statement.
- relop is a relational operator.

If the condition "x relop y" gets satisfied, then-

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

If the condition "x relop y" fails, then-

- The control is not sent to the location specified by label X.
- The next statement appearing in the usual sequence is executed.

4. Unconditional Jump-

```
goto X
```

Here, X is the tag or label of the target statement.

On executing the statement,

- The control is sent directly to the location specified by label X.
- All the statements in between are skipped.

5. Procedure Call-

```
param x call p return y
```

Here, p is a function which takes x as a parameter and returns y.

Example:

1. Write Three Address Code for the following expression-

(a x b) + (c + d) – (a + b + c + d)

Three Address Code for the given expression is-

(1) T1 = a x b

(2) T2 = uminus T1

(3) T3 = c + d

(4) T4 = T2 + T3

(5) T5 = a + b

(6) T6 = T3 + T5

(7) T7 = T4 – T6

2. Write Three Address Code for the following expression

If A < B and C < D then t = 1 else t = 0

(1) If (A < B) goto (3)

(2) goto (4)

(3) If (C < D) goto (6)

(4) t = 0

(5) goto (7)

(6) t = 1

(7)

Program:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#define MAX 100
int tempvar=1;
int addr=100;
void assignmentTAC();
void arithmeticTAC();
void relationalTAC();
void generateTAC(char expr[]);

int main(){
    int choice;
    while(1){
```

```
printf("\n 1 Assignment \n 2. Arithmetic \n 3.
        Relational \n 4. Exit \n 5. Enter your choice:");
scanf("%d", &choice);
switch (choice){
    case 1: assignment TAC(); break;
    case 2: arithmeticTAC(); break;
    case 3: relational TAC(); break;
    case 4: exit(0);
    default: printf("Invalid choice ! Try again \n");
  }
}
}
void assignment TAC(){
  char expr[MAX], lhs[MAX], rhs[MAX];
  printf("\n Enter assignment (eg; a=b): ");
  scanf("%s", expr);
  char *equal sign = strchr(expr, '=');
  if(!equal sign){
      printf("Invalid assignment ! \n");
      return;
  }
  strncpy(lhs, expr, equal sign - expr);
  lhs[equal sign - expr] = '\0';
  strcpy(rhs, equal sign + 1);
  printf("T1 = %s\n", rhs);
  printf("%s", expr);
  generate TAC(expr);
}
```

```
void generate arithmetic TAC(){
    char expr[MAX];
    printf("\n Enter arithmetic expression (eg; a+b+c):");
    scanf("%s", expr);
    generate TAC(expr);
}
void generate TAC (char expr[]){
    char opstack[MAX], operandstack[MAX][MAX];
    int topop = -1, topoperand = -1;
    int i = 0;
    while (expr[i] != '\0'){
        if (isalnum(expr[i])){
            char temp[2] = {expr[i], '\0'};
            strcpy(operandstack[++topoperand], temp);
        }else{
            while (topop != -1 && ((opstack[topop] ==
                '*' || opstack[topop] == '/') ||
                ((expr[i] == '+' || expr[i] == '-') &&
                (opstack[topop] == '+' || opstack[topop] ==
                '-'))){
                char operand2[MAX], operand1[MAX], op;
                strcpy(operand2, operandstack[top
                                    operand--]);
                strcpy(operand1, operandstack[top
                                    operand--]);
                op = opstack[topop--];
                printf("T%d = %s %c %s\n", temp
                        Var, operand1, op, operand2);
```

```
        char tempRes[MAX];
        printf(tempRes, "T%d", tempVar++);
        strcpy(operandstack[++topoperand], tempRes);
      }
      opstack[++topop] = expr[i];
    }
    i++;
  }
  while(topop != -1){
        char operand2[MAX], operand1[MAX], op;
        strcpy(operand2, operandstack[topoperand--]);
        strcpy(operand1, operandstack[topoperand--]);
        op = opstack[topop--];
        printf("T%d = %s %c %s\n", tempVar, operand1,
                                   op, operand2);
        char tempRes[MAX];
        printf(tempRes, "T%d", tempVar++);
        strcpy(operandstack[++topoperand1], tempRes);
      }
  }
  void relationalTAC(){
    char id1[MAX], op[MAX], id2[MAX];
    printf("\n Enter relational expression (eg; A<B):");
    scanf("%s %s %s", op, id2, id1);
    printf("\n%d: if(%s %s %s) goto %d",
            addr, id1, op, id2, addr+3);
    addr++;
}
```

Output

1. Assignment
2. Arithmetic
3. Relational
4. Exit

Enter your choice: 1

Enter assignment (eg; a=b); a=60

$T_1 = 60$

$a = T_1$

VIVA QUESTIONS

1. Define Three Address Code with Example.
2. What is the need for intermediate code?
3. What are the types of three address code?
4. Is three address code machine dependent? Justify.
5. Write the three address code for the following.

   If A < B then 1 else 0

RESULT:

The above program for Three adress code is successfully executed and the Output was verified.

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) (Technical – 5 and Communications - 5)	
Total (100)	

Ex.No.:10 IMPLEMENTATION OF 3-ADDRESS CODE
(TRIPLES, QUADRUPLES AND INDIRECT TRIPLES)

DATE:

AIM

To write a program to implement a code that converts the given expression to triples, quadruples and indirect triples

THEORY:

The commonly used representations for implementing Three Address Code are-

1. Quadruples
2. Triples
3. Indirect Triples

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.

Example:

a + b x c / e ↑ f + b x c

Three Address Code for the given expression is-

$T_1 = e \uparrow f$

$T_2 = b \times c$

$T_3 = T_2 / T_1$

$T_4 = b \times a$

$T_5 = a + T_3$

$T_6 = T_5 + T_4$

Code :-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
type def struct {
        char op[10];
        char arg1[10];
        char arg2[10];
        char result[10];
} Quadruple;
typedef struct {
        char op[10];
        char arg1[10];
        char arg2[10];
} Triple;
type def struct {
    int index;
    int arg1;
    int arg2;
} Indirect Triple;

Quadruple quadruples[MAX];
Triple triples[MAX];
Indirect Triple indirectTriples[MAX];
int qIndex=0, tIndex=0, itIndex=0;
int tempVarCount=1;
```

Program:

```
char stack [MAX][10];
int top = -1;
void push (char *str){
    strcpy (stack [++top], str);
}
char *pop(){
    if (top == -1) return "";
    return stack[top--];
}
int precedence (char op){
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '(') return 3;
    retun 0;
}
void infixToPostfix (char infix[], char postfix[]
                     [10], int * postfixLen){
    char token[10];
    int j=0; k=0;
    for (int i=0; infix[i] != '\0'; i++){
        if (isalnum (infix[i]){
            token[k++] = infix[i];
        } else{
            if (k>0) {
                token [k] = '\0';
                strcpy (postfix[i++], token);
                k=0;
}
```

```
    if (infix[i] == '('){
        push("(");
    } else if (infix[i] == ')'){
        while (top != -1 && strcmp (stack[top], "(" != 0)
        {
            strcpy (postfix[i++], pop());
        } else{
        while (top != -1 && precedence (stack[top]
                [0]) >= precedence (infix[i])){
            strcpy (postfix[j++], pop());
        }
        char op[2] = {infix[i], '\0'};
        push (op);
        }
    }
}
if (k>0){
    token[k] = '\0';
    strcpy (postfix[i], token);
}
while (top != -1){
    strcpy (postfix[j++], pop());
}
*postfixLen = j;
}
void GenerateTempVar (char *temp){
    sprintf (temp, "t%d", tempVarCount++);
}
```

```
Void generateQuadruple(char *op, char *arg1,
                       char *arg2, char *result)
{
  strcpy(quadruples[qIndex].op, op);
  strcpy(quadruples[qIndex].arg1, arg1);
  strcpy(quadruples[qIndex].arg2, arg2);
  strcpy(quadruples[qIndex].result, result);
  qIndex++;
}
Void generateTriple(char *op, char *arg1,
                    char *arg2){
    strcpy(triples[tIndex].op, op);
    strcpy(triples[tIndex].arg1, arg1);
    strcpy(triples[tIndex].arg2, arg2);

    indirectTriples[itIndex].index = tIndex;
    indirectTriples[itIndex].arg1 = (arg1[0]=='t')?
                                    tIndex-1:-1;
    indirectTriples[itIndex].arg2 = (arg2[0]=='t')?
                                    tIndex-1:-1;
    tIndex++;
    itIndex++;
}
Void generateTAC(char postfix[10][], int postfixLen)
    char temp[10];
    for(int i=0; i<postfixLen; i++){
      if(isalnum(postfix[i][0])){
          push(postfix[i]);
      } else{
          char arg2[10], arg1[10], result[10];
```

```
        strcpy (arg2, pop());
        strcpy (arg1, pop());
        generateTempVar (result);
        generate Quadraple (postfix[i], arg1, arg2,
                                                result);
        generate Triple (postfix[i], arg1, arg2);
        push(result);
    }
}
```

Output

Enter arithmetic expression : a=(b*c)+(d*e)

Quadraples:

Index	operator	Arg1	Arg2	result
0	*	b	c	t_1
1	*	d	e	t_2
2	+	t_1	t_2	t_3
3	=	a	t_3	a

Triples:

Index	Operator	Arg1	Arg2
0	*	b	c
1	*	d	e
2	+	t_1	t_2
3	=	a	t_3

VIVA QUESTIONS

1. Compare Triples and Indirect Triples.
2. What are the ways of representation of Intermediate code? (Postfix notation, Syntax tree, Three-address code)
3. State the advantages and disadvantages of Quadruples.
4. Translate the following expression to quadruple, triple and indirect triple-
   a = b x – c + b x – c
5. State the advantages of Indirect Triples.

RESULT:

The above program is executed success fully and the output was verified

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) (Technical – 5 and Communications - 5)	
Total (100)	

Ex .No 11

GENERATION OF TARGET CODE

DATE:

AIM

To write a C program for implementing back end of the compiler which takes three address codes as input and produces 8086 assembly language instruction.

THEORY:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

Code Generator determines the values that are to be stored in the registers.

• For example: consider the three-address statement a := b+c It can have the following sequence of codes:

ADD Rj, Ri Cost = 1

ADD c, Ri Cost = 2

MOV c, Rj Cost = 3

ADD Rj, Ri

Register and Address Descriptors:

• A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

• An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAX 100
void generate TAC(char statement[]);
int main(){
    int n;
    char statements[MAX][MAX];
    printf("Enter the number of statements: ");
    scanf("%d", &n);
    printf("Enter expressions (eg; a=b+c):\n");
    for(int i=0; i<n; i++){
        scanf("%s", statements[i]);
    }
    printf("\n Generated Three Address code (TAC):\n");
    for(int i=0; i<n; i++){
        generate TAC(statements[i]);
    }
    return 0;
}
void generate TAC(char statement[]){
    len = strlen(statement);
    int reg=1;
    char lhs= statement[0];
    char op='\0';
```

```
char operands[2] = {'\0', '\0'};
int opidx = 0;
for (int i=2; i<len; i++){
    if (isalpha(statement[i])){
        operands[opidx++] = statement[i];
        printf("LOAD R%d %c\n", reg++, statement[i]
    } else if (strchr("+*|", statement[i])){
        op = statement[i];
    }
}
if (opidx == 2){
    switch(op){
    case '+': printf("ADD R%d R%d\n",
                reg-2, reg-1"); break;
    case '-': printf("SUB R%d R%d\n", reg-2,
                reg-1); break;
    case '*': printf("MUL R%d R%d\n", reg-2,
                reg-1); break;
    case '/': printf("DIV R%d R%d\n", reg-
                reg-1); break;
    default: printf("Error: unsupported
                operator!\n")
             return;
    }
    printf("STORE %c R%d\n", lhs, reg-2);
}
}
```

OUTPUT:

```
Enter the no of statements2
a=b+c;
c=c+d;
LOAD R1 b
LOAD R2 c
ADD R1 R2
STORE a R1
LOAD R3 c
LOAD R4 d
ADD R3 R4
STORE c R3
```

VIVA QUESTIONS

1. What is the purpose of code generator?
2. List the issues in code generator.
3. Compare Register and Address descriptor.
4. What do you mean by next use information?
5. Write the assembly code for the given expression and find the cost. C=a+b*6
6. Name the technique used for allocating registers efficiently.

   Linear Scan Algorithm

RESULT:

The above program for Generation of Target Code is successfully executed and the Output was Verified.

EVALUATION

Assessment	Marks Scored
Understanding Problem statement (10)	
Efficiency of understanding algorithm (20)	
Efficiency of program (40)	
Output (20)	
Viva (10) (Technical – 5 and Communications - 5)	
Total (100)	

Ex .No 12 IMPLEMENTATIONS OF OPTIMIZATION TECHNIQUES

DATE:

AIM

To write a program to implement a code optimizer to perform possible optimization like dead code elimination, common sub expression elimination, etc,.

THEORY:

Reasons for Optimizing the Code

- Code optimization is essential to enhance the execution and efficiency of a source code.
- It is mandatory to deliver efficient target code by lowering the number of instructions in a program.

When to Optimize?

Code optimization is an important step that is usually performed at the last stage of development.

Role of Code Optimization

- It is the fifth stage of a compiler, and it allows you to choose whether or not to optimize your code, making it really optional.
- It aids in reducing the storage space and increases compilation speed.
- It takes source code as input and attempts to produce optimal code.
- Functioning the optimization is tedious; it is preferable to employ a code optimizer to accomplish the assignment.

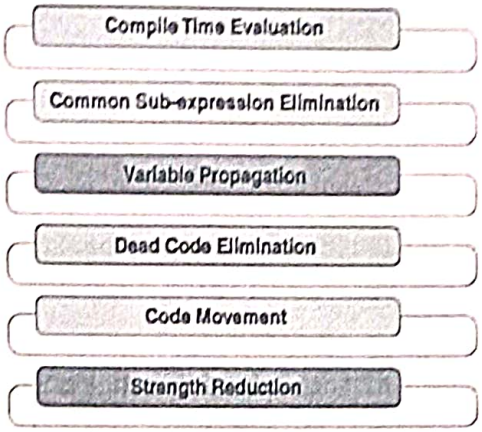
Different Types of Optimization

Optimization is classified broadly into two types:

- Machine-Independent
- Machine-Dependent

Machine-Independent Optimization

It positively affects the efficiency of intermediate code by transforming a part of code that does not employ hardware parts. It usually optimises code by eliminating tediums and removing unneeded code.



Machine-Dependent Optimization

After the target code has been constructed and transformed according to the target machine architecture, machine-dependent optimization is performed. It makes use of CPU registers and may utilise absolute rather than relative memory addresses. Machine-dependent optimizers work hard to maximise the perks of the memory hierarchy.

Loop Optimization

- Invariant code/Code Motion or Frequency Reduction
- Induction analysis
- Strength reduction

ALGORITHM:

1. Start the program
2. Get the coding from the user
3. Find the operators, arguments and results from the coding
4. Display the value in the table.
5. Stop the program

PROGRAM:

```
#include<stdio.h>
#include<string.h>
struct op {
    char l;
    char r[20];
} op[10], pr[10];
int main(){
    int a, i, k, j, n, z=0, m, q;
    char *p, *l;
    char * temp;
    printf("Enter number of values:");
    scanf("%d", &n);
    for (i=0; i<n; i++){
        printf("Left:");
        scanf("%c", &op[i].l);
        printf("Right:");
        scanf("%s", op[i].r);
    }
    printf("\n Intermediate Code\n");
    for(i=0; i<n; i++) {
        printf("%c = %s\n", op[i].l, op[i].r);
    }
    for(i=0; i<n-1; i++){
        temp = op[i];
        for(j=0; j<n; j++){
            p= strchr(op[j].r, temp);
```

```
        if (p){
            pr[z].l = op[i].l;
            strcpy(pr[z].r, op[i].r);
            z++;
            break;
        }
    }
}
pr[z].l = op[n-1].l;
strcpy(pr[z].r, op[n-1].r);
z++;
printf("\n After Dead code Elimination: \n");
for(k=0; k<z; k++){
    printf("%c = %s\n", pr[k].l, pr[k].r);
}
for(m=0; m<z; m++){
    tem = pr[m].r;
    for(j=m+1; j<z; j++){
        p = strstr(tem, pr[j].r);
        if(p){
            t = pr[j].l
            pr[j].l = pr[m].l;
            for(i=0; i<z; i++){
                l = strchr(pr[i].r, t);
                if(l){
                    a = l - p[i].r;
                    p[i].r[a] = pr[m].l;
                }
            }
        }
    }
}
```

```
printf("\n After Eliminating Common Sub expression:\n");
for(i=0; i<z; i++){
    printf("%c = %s\n", pr[i].l, pr[i].r);
}
for(i=0; i<z; i++){
    for(j=i+1; j<z; j++){
        q = strcmp(pr[i].r, pr[j].r);
        if((pr[i].l == pr[j].l) && q==0){
            pr[j].l = '\0';
            strcpy(pr[j].r, "");
        }
    }
}
printf("\n Optimized code :\n");
for(i=0; i<z; i++){
    if(pr[i].l != '\0'){
        printf("%c = %s\n", pr[i].l, pr[i].r);
    }
}
return 0;
}
```

OUTPUT:

```
enter no of values 5
left   a right:  9
left   b right: c+d
left   e right: c+d
left   f right: b+e
left   r right: f
intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f

after dead code elimination
b     =c+d
e     =c+d
f     =b+e
r     =f
eliminate common expression
b     =c+d
b     =c+d
f     =b+b
r     =f
optimized code
b=c+d
f=b+b
r=f
```

RESULT:

The above program is executed Successfully and the output was verified.