Thadomal Shahani Engineering College

Bandra (W.), Mumbai - 400 050.

CERTIFICATE

Certify that Mr./Miss Aaman Malvia

of Computer Department, Semester VI with

Roll No. 2103109 has completed a course of the necessary

experiments in the subject SPCC under my

supervision in the **Thadomal Shahani Engineering College**

Laboratory in the year 2023 - 2024

8/4/24

Teacher In- Charge

Head of the Department

Date 8/4/24

Principal

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

CONTENTS

SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

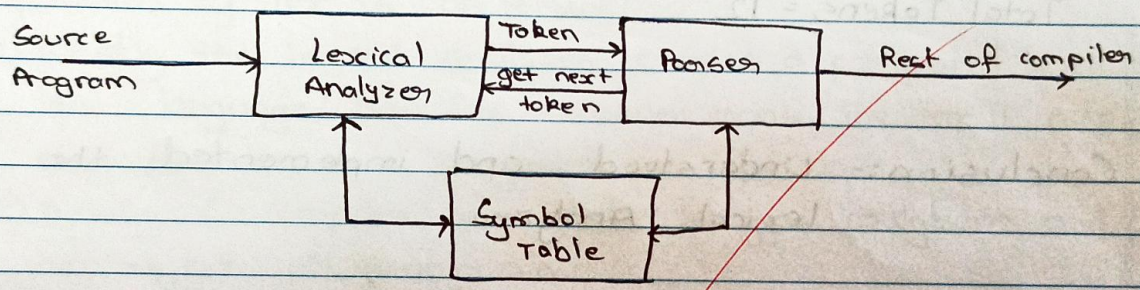
Arnav Malvia
2103109
C23

Experiment :- 1

Aim:- Write a program to implement Lexical Analyzer.

Theory:-

The lexical analyzer is responsible for removing the white spaces and comments from the source program. It corresponds to the error messages with the source program. It helps to identify the tokens. The input characters are read by the lexical analyzer from the source code.



Working:-

1) Input Preprocessing- This stage involves cleaning up the input and preparing it for lexical analysis. This may include removing comments, whitespaces and other non-essential characters.
2) Tokenisation - This is the process of breaking the input text into a sequence of tokens.
3) Token classification:- In this stage the lexer determines the type of each token.
4) Token validation - In this stage, the lexer checks that each token is valid according to the rules of programming language.
5) Output Generation- This is the final stage, the lexer generates

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

eg:- ~~a b * d~~ " a = b * 4 + (8 * 4) ; "

Keywords = 1
Characters = 2
Operators = 4
Seperators = 2
Numbers = 3
Total Tokens = 12

Conclusion:- Understood and implemented the lexical Analyzer..

23/11/24 A+

Experiment 1:-

Code:

```
// C program to illustrate the implementation of lexical analyser

#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LENGTH 100

// this function check for a delimiter(it is a piece of data
// that seprated it from other) to peform some specif case
// on it
bool isDelimiter(char chr)
{
	return (chr == ' ' || chr == '+' || chr == '-'
				|| chr == '*' || chr == '/' || chr == ','
				|| chr == ';' || chr == '%' || chr == '>'
				|| chr == '<' || chr == '=' || chr == '('
				|| chr == ')' || chr == '[' || chr == ']'
				|| chr == '{' || chr == '}');
}

// this function check for a valid identifier eg:- +,-* etc
bool isOperator(char chr)
{
	return (chr == '+' || chr == '-' || chr == '*'
```

```
                || chr == '/' || chr == '>' || chr == '<'
                || chr == '=');
}

// this function check for an valid identifier
bool isValidIdentifier(char* str)
{
        return (str[0] != '0' && str[0] != '1' && str[0] != '2'
                && str[0] != '3' && str[0] != '4'
                && str[0] != '5' && str[0] != '6'
                && str[0] != '7' && str[0] != '8'
                && str[0] != '9' && !isDelimiter(str[0]));
}

// 32 Keywords are checked in this function and return the
// result accordingly
bool isKeyword(char* str)
{
        const char* keywords[]
                = { "auto",     "break", "case",       "char",
                    "const", "continue", "default", "do",
                    "double", "else",       "enum",       "extern",
                    "float", "for",  "goto",          "if",
                    "int",    "long",          "register", "return",
                    "short", "signed", "sizeof", "static",
                    "struct", "switch", "typedef", "union",
                    "unsigned", "void",   "volatile", "while" };
        for (int i = 0;
                i < sizeof(keywords) / sizeof(keywords[0]); i++) {
```

```
            if (strcmp(str, keywords[i]) == 0) {
                return true;
            }
        }
        return false;
}

// check for an integer value
bool isInteger(char* str)
{
        if (str == NULL || *str == '\0') {
            return false;
        }
        int i = 0;
        while (isdigit(str[i])) {
            i++;
        }
        return str[i] == '\0';
}

// trims a substring from a given string's start and end
// position
char* getSubstring(char* str, int start, int end)
{
        int length = strlen(str);
        int subLength = end - start + 1;
        char* subStr
            = (char*)malloc((subLength + 1) * sizeof(char));
        strncpy(subStr, str + start, subLength);
```

```
        subStr[subLength] = '\0';
        return subStr;
}

// this function parse the input
int lexicalAnalyzer(char* input)
{
        int left = 0, right = 0;
        int len = strlen(input);

        while (right <= len && left <= right) {
                if (!isDelimiter(input[right]))
                        right++;

                if (isDelimiter(input[right]) && left == right) {
                        if (isOperator(input[right]))
                                printf("Token: Operator, Value: %c\n",
                                        input[right]);

                        right++;
                        left = right;
                }
                else if (isDelimiter(input[right]) && left != right
                                || (right == len && left != right)) {
                        char* subStr
                                = getSubstring(input, left, right - 1);

                        if (isKeyword(subStr))
                                printf("Token: Keyword, Value: %s\n",
```

```
                        subStr);

                else if (isInteger(subStr))
                        printf("Token: Integer, Value: %s\n",
                                subStr);

                else if (isValidIdentifier(subStr)
                                && !isDelimiter(input[right - 1]))
                        printf("Token: Identifier, Value: %s\n",
                                subStr);

                else if (!isValidIdentifier(subStr)
                                && !isDelimiter(input[right - 1]))
                        printf("Token: Unidentified, Value: %s\n",
                                subStr);
                left = right;
            }
        }
        return 0;
}

// main function
int main()
{
        // Input 01
        char lex_input[MAX_LENGTH] = "int a = b + c";
        printf("For Expression \"%s\":\n", lex_input);
        lexicalAnalyzer(lex_input);
        printf(" \n");
```

```
    // Input 02
    char lex_input01[MAX_LENGTH]
        = "int x=ab+bc+30+switch+ 0y ";
    printf("For Expression \"%s\":\n", lex_input01);
    lexicalAnalyzer(lex_input01);
    return (0);
}
```

Output:

```
For Expression "int a = b + c":
Token: Keyword, Value: int
Token: Identifier, Value: a
Token: Operator, Value: =
Token: Identifier, Value: b
Token: Operator, Value: +
Token: Identifier, Value: c

For Expression "int x=ab+bc+30+switch+ 0y ":
Token: Keyword, Value: int
Token: Identifier, Value: x
Token: Operator, Value: =
Token: Identifier, Value: ab
Token: Operator, Value: +
Token: Identifier, Value: bc
Token: Operator, Value: +
Token: Integer, Value: 30
Token: Operator, Value: +
Token: Keyword, Value: switch
Token: Operator, Value: +
Token: Unidentified, Value: 0y
```

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

SPCC
Aarnav
Maluia
2103109
C23

Experiment :- 2

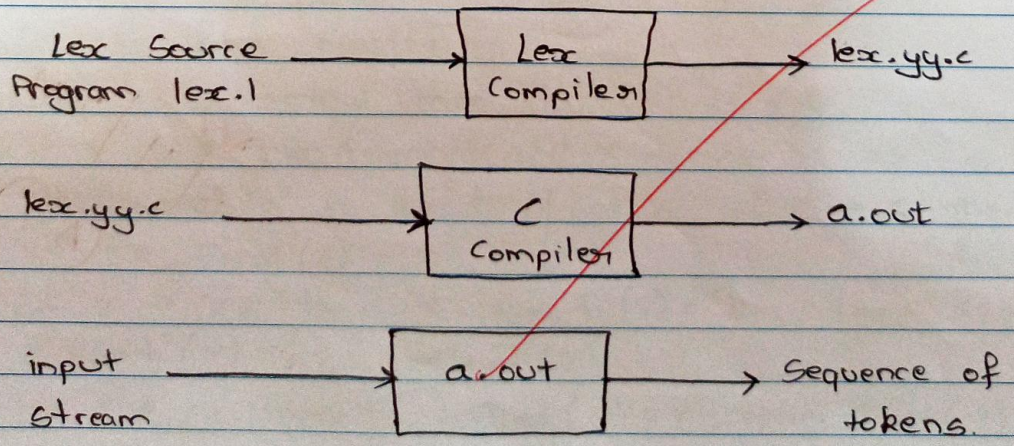
Aim :- To study and implement programs LEX and YACC tool.

Theory :-

Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Function of lex is as follows :-

1) Firstly the lexical analyzer creates a program lex.l in the lex language. Then Lex compiler runs the lex.l program and produces a C program lex.yy.c.
2) Finally C compiler runs the lex.yy.c program and produces an object program a.out.
3) a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

For Compiling a YACC program:-

1) Write a lex program in a file file.l and yacc in a file file.y
2) Open terminal and navigate to the directory where you have saved the files.
3) type lex file.l
4) type yacc file.y
5) type cc lex.yy.c y.tab.h -ll
6) type ./a.out

Conclusion:- Understood and implemented LEX and YACC tools.

10/10

Experiment 2 :-

prog1:

```
%{


%}


%%
\n { printf("Hello World !");}


%%


void main()
{
yylex();
}
```

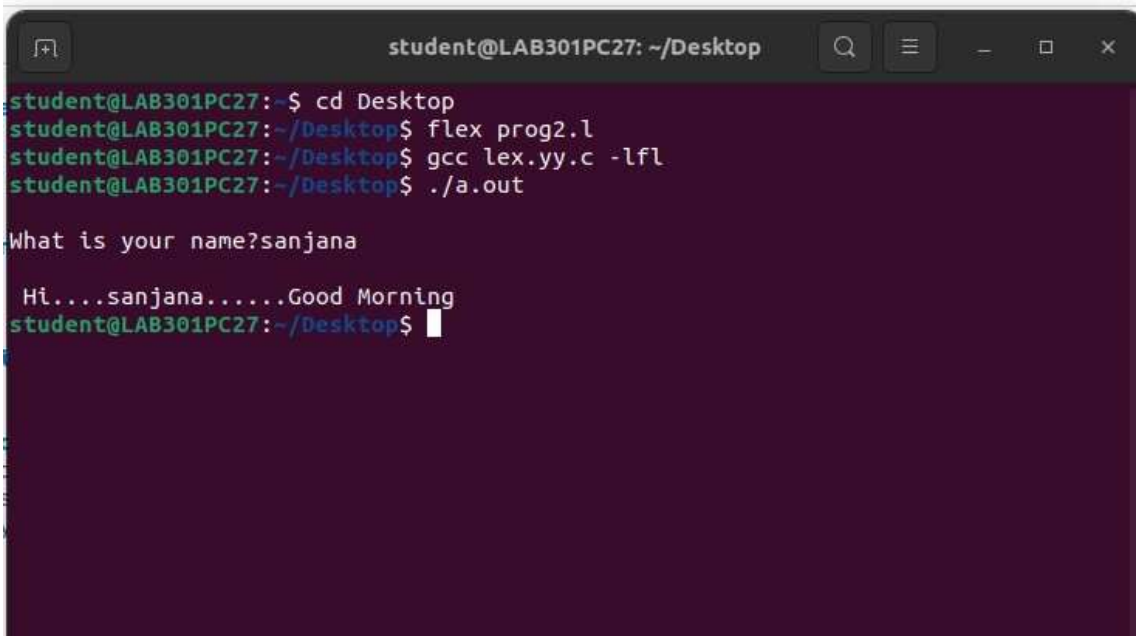
Output:

prog2:

```
 %{
char name[10];
%}

%%

[\n] {printf("\n Hi....%s.....Good Morning\n",name);return 1;}

%%

void main()
{
```

```
  char opt;
  printf("\nWhat is your name?");
  scanf("%s",name);
  yylex();
}
```

Output:

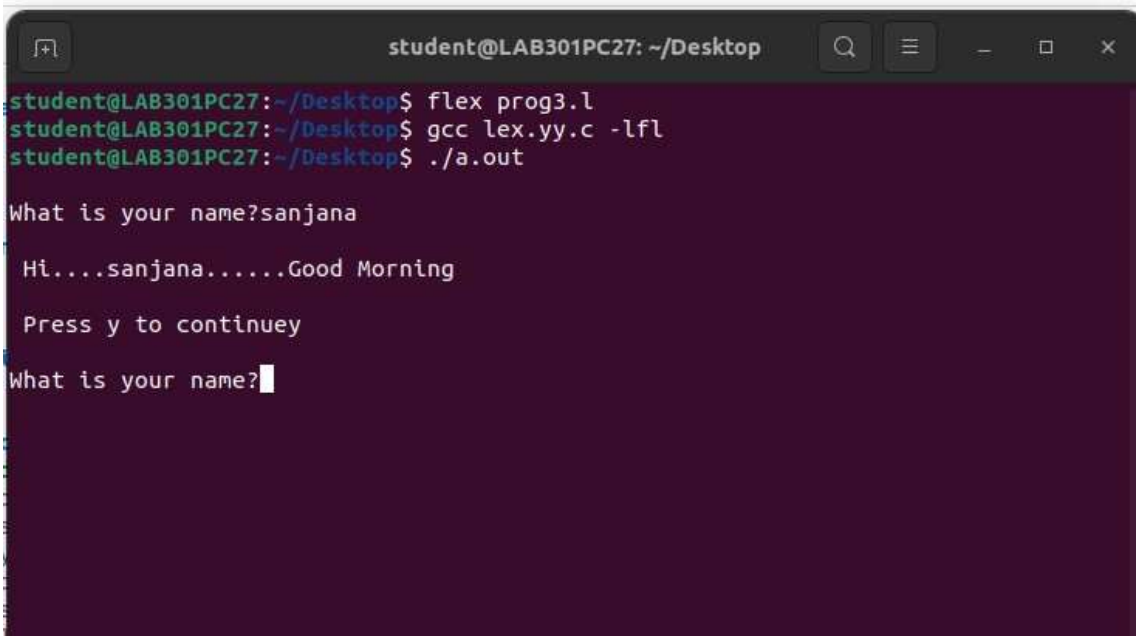


prog3:

```
%{
char name[10];
%}

%%

[\n] {printf("\n Hi....%s.....Good Morning\n",name);return 1;}

%%
```

```
void main()
{

  char opt;
  do {
  printf("\nWhat is your name?");
  scanf("%s",name);
  yylex();
  printf("\n Press y to continue");
  scanf("%c",&opt);
  }
  while(opt=='y');
}
```

Output:

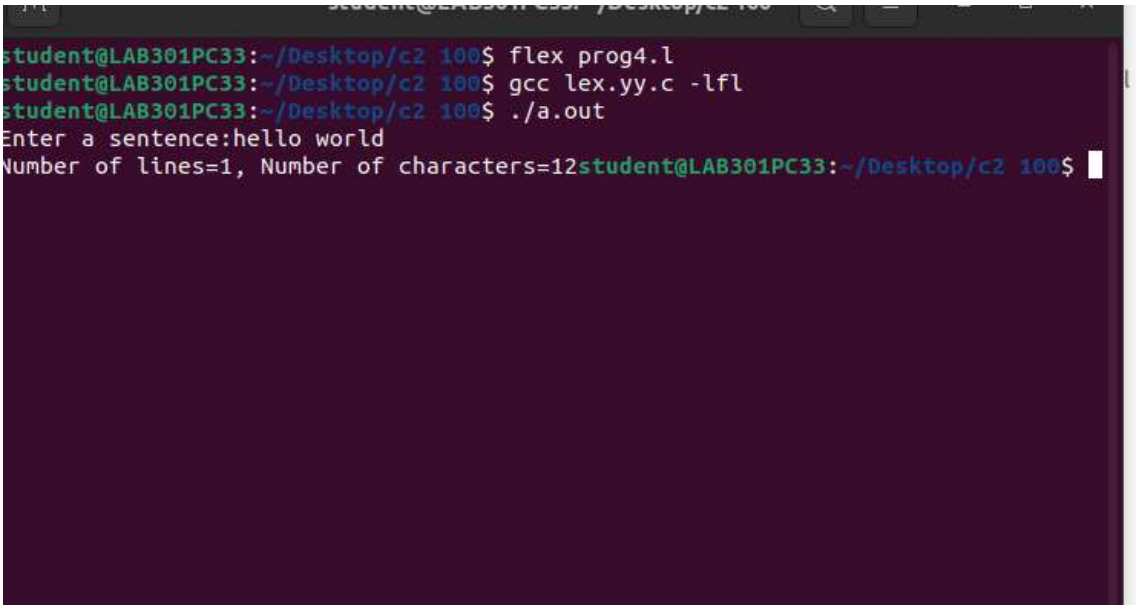


prog 4:

```
%{
int linecount=0, charcount=0, sc=0, tc=0, wc=0;
```

```
%}
%%
[\n]     {linecount++; charcount+=yyleng;}
[\t]     {sc++; charcount+=yyleng;}
[^\t]    {tc++; charcount+=yyleng;}
[^\t\n]+         {wc++; charcount+=yyleng;}
%%

int yywrap() { return 1; }

void main()
{
        printf("Enter a sentence:");
        yylex();
        printf("Number of lines=%d, Number of characters=%d",linecount, charcount);
}
```

Output:



prog 5:

```
void display(int, char *);
 int flag;
%}
%%
[a|e|i|o|u] { flag=1; display(flag,yytext); }
            {flag = 0; display(flag,yytext); }
%%
void main()
{
 printf("\nEnter the word:"); yylex();
}
void display(int flag, char *t);
{
  if(flag==1)
  {
   printf("\nThe given character %s is vowel\n",t);
  }
  else
  {
  printf("\nThe given character %s is not vowel\n",t);
  }
}
```

Output:

```
enter the word:tsec

the given character t is not vowel

the given character s is not vowel

the given character e is vowel

the given character c is not vowel
```

CDIT
SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Arnav Maduia
2103109
C23

Experiment :- 3

Aim:- Write a program to implement the FIRST and FOLLOW set for the given grammar.

Theory:-

First and Follow in compiler design are two grammatical functions that help you enter table entries. We will discuss the First and Follow in detail below. If the compiler knew ahead of time what the "initial character and follow up of the string produced when a production rule is applied," it might carefully choose which production rule to apply by comparing it to the current character or token in the input string it sees.

~~FIR~~ First ():

First () is a function that specifies the set of terminals that start a string derived from a production rule. It is the first terminal that appears on the right-hand side of the production. ~~xxxx~~

Rules to find First():-

1) If x is a terminal, then First(x) is {x}
2) If x is a non-terminal and x tends to ax is production, then add 'a' to the first of x. if $X \to \epsilon$ then add null to the First (x).
3) If $X \to YZ$ then if First (Y) = ϵ, then First(x) = { First (Y) - ϵ } $\cup$ First (Z).
4) If $X \to YZ$ then if First (x) = Y, then First(Y) = terminal but null then First(x) = First(Y) = terminals

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Follow():-

Follow() is a set of terminals symbols that can be displayed just to the right of the non-terminal symbol in any sentence format. It is the first non-terminal appearing after the given terminal symbol on the right hand side of production.

Rules to find Follow():-

1) $ is a follow of 'S' (Start Symbol)
2) If $A \rightarrow \alpha B\beta$, $\beta != \epsilon$ then first(β) is in follow(B)
3) If $A \rightarrow aB$ or $A \rightarrow \alpha B\beta$ where First(β) = ϵ, then everything in Follow(A) is a Follow(B)

Conclusion:- Understood and implemented the FIRST and FOLLOW set for the given grammar program.

27/2/14 A+

Experiment 3 :-

Code:

```
def computeFirst(rule, rules, nonterm_userdef, term_userdef, diction,
firsts):
    if len(rule) != 0 and (rule is not None):
        if rule[0] in term_userdef:
            return [rule[0]]
        elif rule[0] == '#':
            return ['#']
        if len(rule) != 0:
            if rule[0] in list(diction.keys()):
                fres = []
                rhs_rules = diction[rule[0]]
                for itr in rhs_rules:
                    indivRes = computeFirst(itr, rules, nonterm_userdef,
                                            term_userdef, diction, firsts)
                    if type(indivRes) is list:
                        fres.extend(indivRes)
                    else:
                        fres.append(indivRes)
                if '#' not in fres:
                    return list(set(fres))
                else:
                    fres.remove('#')
            if len(rule) > 1:
                ansNew = computeFirst(rule[1:], rules, nonterm_userdef,
                                      term_userdef, diction, firsts)
                if ansNew is not None:
                    if type(ansNew) is list:
                        return list(set(fres).union(set(ansNew)))
                    else:
                        return list(set(fres).union({ansNew}))
            fres.append('#')
        return list(set(fres))

def computeFollow(nt, start_symbol, rules, nonterm_userdef, term_userdef,
                  diction, firsts, follows):
    solset = set()
    res = set()  # Initialize res with an empty set
    if nt == start_symbol:
        solset.add('$')
    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            if nt in subrule:
                index_nt = subrule.index(nt)
                remaining_subrule = subrule[index_nt + 1:]
                if len(remaining_subrule) != 0:
                    res = computeFirst(remaining_subrule, rules,
                                       nonterm_userdef, term_userdef,
                                       diction, firsts)
                    if '#' in res:
                        res.remove('#')
                    ansNew = computeFollow(curNT, start_symbol, rules,
                                           nonterm_userdef, term_userdef,
                                           diction, firsts, follows)
                    if ansNew is not None:
                        solset.update(res)
                        solset.update(ansNew)
```

```
                    else:
                        solset.update(res)
                else:
                    solset.update(res)
                    if len(remaining_subrule) == 0 or '#' in \
                            computeFirst(remaining_subrule, rules,
                                         nonterm_userdef, term_userdef,
                                         diction, firsts):
                        if nt != curNT:
                            ansNew = computeFollow(curNT, start_symbol,
                                                   rules, nonterm_userdef,
                                                   term_userdef, diction,
                                                   firsts, follows)
                            if ansNew is not None:
                                solset.update(ansNew)
    return list(solset)

# Example usage
rules = ["S -> A S | C",
         "A -> a | b | B c",
         "B -> p | a",
         "C -> c "]
nonterm_userdef = ['S', 'A', 'B', 'C']
term_userdef = ['a', 'c', 'b', 'p']

# Diction, Firsts, Follows initialization
diction = {}
firsts = {}
follows = {}

# computeFirst for each rule
for rule in rules:
    k = rule.split("->")
    k[0] = k[0].strip()
    k[1] = k[1].strip()
    rhs = k[1]
    multirhs = rhs.split('|')
    for i in range(len(multirhs)):
        multirhs[i] = multirhs[i].strip()
        multirhs[i] = multirhs[i].split()
    diction[k[0]] = multirhs

# computeFirst for each non-terminal
for y in list(diction.keys()):
    t = set()
    for sub in diction.get(y):
        res = computeFirst(sub, rules, nonterm_userdef, term_userdef,
diction,
                           firsts)
        if res is not None:
            if type(res) is list:
                for u in res:
                    t.add(u)
            else:
                t.add(res)
    firsts[y] = t

# computeFollow for each non-terminal
for NT in diction:
    solset = set()
    sol = computeFollow(NT, list(diction.keys())[0], rules,
```

```
nonterm_userdef,
                        term_userdef, diction, firsts, follows)
    if sol is not None:
        for g in sol:
            solset.add(g)
    follows[NT] = solset

# Print the results
print("\nCalculated firsts: ")
key_list = list(firsts.keys())
index = 0
for gg in firsts:
    print(f"first({key_list[index]}) => {firsts.get(gg)}")
    index += 1

print("\nCalculated follows: ")
key_list = list(follows.keys())
index = 0
for gg in follows:
    print(f"follow({key_list[index]}) => {follows[gg]}")
    index += 1
```

Output:

SPCC
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
Arnav Malvia
C23
2103109

Experiment :- 4

Aim:- Write a program to implement Parser.

Theory:-

- LL(1) grammar is used to construct predictive parser. It is a recursive descent parser with the need of Backtracking.

- First L stands for scanning input from left to right. Second L stands for Leftmost derivation. 1 stands for one input symbol of look ahead each step to make parsing action decisions.

Example:

$E \rightarrow TE'$ $\quad T' \rightarrow *FT' \mid \epsilon$

$E' \rightarrow +TE' \mid \epsilon$ $\quad F \rightarrow (E) \mid id$

$T \rightarrow FT'$

- Parsing table has non terminals as rows and terminals as columns.

- **Step 1:**

For each terminal "a" in FIRST(α)

ADD $A \rightarrow \alpha$ to $[A, \alpha]$

Step 2:

Case 1: If ϵ is first(α) then for each terminal b in FOLLOW(A)

ADD $A \rightarrow \alpha$ to $[A, b]$

Case 2: If ϵ in first(α) and $ in follow(A)

then for each terminal b in Follow(A)

ADD $A \rightarrow \alpha$ to $M[A, b]$

Step 3:-
Make each undefined entry of M be an error.

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$T \to (E)$		

Applying predictive passing algorithm to get:-
id + id * id

→ Matched	Stack	Input	Action
	E $	id+id*id $	
	TE' $	id+id*id $	output $E' \to TE'$
	FTE' $	id+id*id $	$T \to FT'$
	idT'E' $	id+id*id $	$F \to id$
id	T'E' $	+id*id $	match id
id	→ TE' $	+id*id $	$T' \to \epsilon$
id	TE' $	+id*id $	$E' \to +TE'$
id+	FT'E $	id*id $	match +
id+●	idT'E' $	id*id $	$T \to FT'$
id+id	T'E' $	id*id $	$F \to id$
id+id●	FT'E' $	*id $	match id
id+id*	FT'E' $	*id $	output $T' \to FT'$
id+id*●	idT'E' $	id $	match *
id+id*id	T'E' $	id $	$F \to id$
id+id*id	E' $	$	match id
id+id*id	$	$	$T' \to \epsilon$

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Conclusion: Understood and Implemented parser.

19/3/24

A+

Experiment 4 :-

LL1:

```
# LL(1) parser code in python


def removeLeftRecursion(rulesDiction):
   # for rule: A->Aa|b
   # result: A->bA',A'->aA'|#


   # 'store' has new rules to be added
   store = {}
   # traverse over rules
   for lhs in rulesDiction:
      # alphaRules stores subrules with left-recursion
      # betaRules stores subrules without left-recursion
      alphaRules = []
      betaRules = []
      # get rhs for current lhs
      allrhs = rulesDiction[lhs]
      for subrhs in allrhs:
         if subrhs[0] == lhs:
            alphaRules.append(subrhs[1:])
         else:
            betaRules.append(subrhs)
      # alpha and beta containing subrules are separated
      # now form two new rules
      if len(alphaRules) != 0:
         # to generate new unique symbol
         # add ' till unique not generated
         lhs_ = lhs + "'"
         while (lhs_ in rulesDiction.keys()) \
               or (lhs_ in store.keys()):
            lhs_ += "'"
         # make beta rule
         for b in range(0, len(betaRules)):
            betaRules[b].append(lhs_)
         rulesDiction[lhs] = betaRules
         # make alpha rule
         for a in range(0, len(alphaRules)):
            alphaRules[a].append(lhs_)
         alphaRules.append(['#'])
         # store in temp dict, append to
         # - rulesDiction at end of traversal
         store[lhs_] = alphaRules
```

```
   # add newly generated rules generated
   # - after removing left recursion
   for left in store:
      rulesDiction[left] = store[left]
   return rulesDiction
```

```
def LeftFactoring(rulesDiction):
   # for rule: A->aDF|aCV|k
   # result: A->aA'|k, A'->DF|CV


   # newDict stores newly generated
   # - rules after left factoring
   newDict = {}
   # iterate over all rules of dictionary
   for lhs in rulesDiction:
      # get rhs for given lhs
      allrhs = rulesDiction[lhs]
      # temp dictionary helps detect left factoring
      temp = dict()
      for subrhs in allrhs:
         if subrhs[0] not in list(temp.keys()):
            temp[subrhs[0]] = [subrhs]
         else:
            temp[subrhs[0]].append(subrhs)
      # if value list count for any key in temp is > 1,
      # - it has left factoring
      # new_rule stores new subrules for current LHS symbol
      new_rule = []
      # temp_dict stores new subrules for left factoring
      tempo_dict = {}
      for term_key in temp:
         # get value from temp for term_key
         allStartingWithTermKey = temp[term_key]
         if len(allStartingWithTermKey) > 1:
            # left factoring required
            # to generate new unique symbol
            # - add ' till unique not generated
            lhs_ = lhs + "'"
            while (lhs_ in rulesDiction.keys()) \
                  or (lhs_ in tempo_dict.keys()):
               lhs_ += "'"
```

```
            # append the left factored result
            new_rule.append([term_key, lhs_])
            # add expanded rules to tempo_dict
            ex_rules = []
            for g in temp[term_key]:
               ex_rules.append(g[1:])
            tempo_dict[lhs_] = ex_rules
         else:
            # no left factoring required
            new_rule.append(allStartingWithTermKey[0])
      # add original rule
      newDict[lhs] = new_rule
      # add newly generated rules after left factoring
```

```
      for key in tempo_dict:
         newDict[key] = tempo_dict[key]
   return newDict


# calculation of first
# epsilon is denoted by '#' (semi-colon)


# pass rule in first function
def first(rule):
   global rules, nonterm_userdef, \
      term_userdef, diction, firsts
   # recursion base condition
   # (for terminal or epsilon)
   if len(rule) != 0 and (rule is not None):
      if rule[0] in term_userdef:
         return rule[0]
      elif rule[0] == '#':
         return '#'

   # condition for Non-Terminals
   if len(rule) != 0:
      if rule[0] in list(diction.keys()):
         # fres temporary list of result
         fres = []
         rhs_rules = diction[rule[0]]
         # call first on each rule of RHS
         # fetched (& take union)
         for itr in rhs_rules:
            indivRes = first(itr)
```

```
            if type(indivRes) is list:
               for i in indivRes:
                  fres.append(i)
            else:
               fres.append(indivRes)


         # if no epsilon in result
         # - received return fres
         if '#' not in fres:
            return fres
         else:
            # apply epsilon
            # rule => f(ABC)=f(A)-{e} U f(BC)
            newList = []
            fres.remove('#')
            if len(rule) > 1:
               ansNew = first(rule[1:])
               if ansNew != None:
                  if type(ansNew) is list:
```

```
                     newList = fres + ansNew
                  else:
                     newList = fres + [ansNew]
               else:
                  newList = fres
               return newList
            # if result is not already returned
            # - control reaches here
            # lastly if eplison still persists
            # - keep it in result of first
            fres.append('#')
            return fres


# calculation of follow
# use 'rules' list, and 'diction' dict from above


# follow function input is the split result on
# - Non-Terminal whose Follow we want to compute
def follow(nt):
   global start_symbol, rules, nonterm_userdef, \
      term_userdef, diction, firsts, follows
   # for start symbol return $ (recursion base case)
```

```
solset = set()
if nt == start_symbol:
    # return '$'
    solset.add('$')


# check all occurrences
# solset - is result of computed 'follow' so far


# For input, check in all rules
for curNT in diction:
    rhs = diction[curNT]
    # go for all productions of NT
    for subrule in rhs:
        if nt in subrule:
            # call for all occurrences on
            # - non-terminal in subrule
            while nt in subrule:
                index_nt = subrule.index(nt)
                subrule = subrule[index_nt + 1:]
                # empty condition - call follow on LHS
                if len(subrule) != 0:
                    # compute first if symbols on
                    # - RHS of target Non-Terminal exists
                    res = first(subrule)
                    # if epsilon in result apply rule
```

```
                    # - (A->aBX)- follow of -
                    # - follow(B)=(first(X)-{ep}) U follow(A)
                    if '#' in res:
                        newList = []
                        res.remove('#')
                        ansNew = follow(curNT)
                        if ansNew != None:
                            if type(ansNew) is list:
                                newList = res + ansNew
                            else:
                                newList = res + [ansNew]
                        else:
                            newList = res
                        res = newList
                else:
                    # when nothing in RHS, go circular
```

```
                # - and take follow of LHS
                # only if (NT in LHS)!=curNT
                if nt != curNT:
                    res = follow(curNT)


            # add follow result in set form
            if res is not None:
                if type(res) is list:
                    for g in res:
                        solset.add(g)
                else:
                    solset.add(res)
    return list(solset)


def computeAllFirsts():
    global rules, nonterm_userdef, \
        term_userdef, diction, firsts
    for rule in rules:
        k = rule.split("->")
        # remove un-necessary spaces
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        # remove un-necessary spaces
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs


    print(f"\nRules: \n")
    for y in diction:
```

```
        print(f"{y}->{diction[y]}")
    print(f"\nAfter elimination of left recursion:\n")


    diction = removeLeftRecursion(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")
    print("\nAfter left factoring:\n")
```

```
    diction = LeftFactoring(diction)
    for y in diction:
        print(f"{y}->{diction[y]}")


    # calculate first for each rule
    # - (call first() on all RHS)
    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)


        # save result in 'firsts' list
        firsts[y] = t


    print("\nCalculated firsts: ")
    key_list = list(firsts.keys())
    index = 0
    for gg in firsts:
        print(f"first({key_list[index]}) "
            f"=> {firsts.get(gg)}")
        index += 1


def computeAllFollows():
    global start_symbol, rules, nonterm_userdef,\
        term_userdef, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            for g in sol:
                solset.add(g)
        follows[NT] = solset


    print("\nCalculated follows: ")
```

```
    key_list = list(follows.keys())
    index = 0
    for gg in follows:
        print(f"follow({key_list[index]})"
            f" => {follows[gg]}")
        index += 1


# create parse table
def createParseTable():
    import copy
    global diction, firsts, follows, term_userdef
    print("\nFirsts and Follow Result table\n")


    # find space size
    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2


    print(f"{{:<{10}}} "
        f"{{:<{mx_len_first + 5}}} "
        f"{{:<{mx_len_fol + 5}}}"
        .format("Non-T", "FIRST", "FOLLOW"))
    for u in diction:
        print(f"{{:<{10}}} "
            f"{{:<{mx_len_first + 5}}} "
            f"{{:<{mx_len_fol + 5}}}"
            .format(u, str(firsts[u]), str(follows[u])))


    # create matrix of row(NT) x [col(T) + 1($)]
    # create list of non-terminals
    ntlist = list(diction.keys())
    terminals = copy.deepcopy(term_userdef)
    terminals.append('$')


    # create the initial empty state of ,matrix
```

```
mat = []
for x in diction:
    row = []
    for y in terminals:
        row.append('')
    # of $ append one more col
    mat.append(row)
```

```


# Classifying grammar as LL(1) or not LL(1)
grammar_is_LL = True


# rules implementation
for lhs in diction:
    rhs = diction[lhs]
    for y in rhs:
        res = first(y)
        # epsilon is present,
        # - take union with follow
        if '#' in res:
            if type(res) == str:
                firstFollow = []
                fol_op = follows[lhs]
                if fol_op is str:
                    firstFollow.append(fol_op)
                else:
                    for u in fol_op:
                        firstFollow.append(u)
                res = firstFollow
            else:
                res.remove('#')
                res = list(res) +\
                    list(follows[lhs])
        # add rules to table
        ttemp = []
        if type(res) is str:
            ttemp.append(res)
            res = copy.deepcopy(ttemp)
        for c in res:
            xnt = ntlist.index(lhs)
            yt = terminals.index(c)
            if mat[xnt][yt] == '':
                mat[xnt][yt] = mat[xnt][yt] \
```

```
                        + f"{lhs}->{' '.join(y)}"
            else:
                # if rule already present
                if f"{lhs}->{y}" in mat[xnt][yt]:
                    continue
                else:
                    grammar_is_LL = False
                    mat[xnt][yt] = mat[xnt][yt] \
                            + f",{lhs}->{' '.join(y)}"


    # final state of parse table
    print("\nGenerated parsing table:\n")
    frmt = "{:>12}" * len(terminals)
    print(frmt.format(*terminals))
```

```
    j = 0
    for y in mat:
        frmt1 = "{:>12}" * len(y)
        print(f"{ntlist[j]} {frmt1.format(*y)}")
        j += 1


    return (mat, grammar_is_LL, terminals)


def validateStringUsingStackBuffer(parsing_table, grammarll1,
                        table_term_list, input_string,
                        term_userdef,start_symbol):


    print(f"\nValidate String => {input_string}\n")


    # for more than one entries
    # - in one cell of parsing table
    if grammarll1 == False:
        return f"\nInput String = " \
            f"\"{input_string}\"\n" \
            f"Grammar is not LL(1)"


    # implementing stack buffer
```

```
stack = [start_symbol, '$']
buffer = []


# reverse input string store in buffer
input_string = input_string.split()
input_string.reverse()
buffer = ['$'] + input_string


print("{:>20} {:>20} {:>20}".
   format("Buffer", "Stack","Action"))


while True:
   # end loop if all symbols matched
   if stack == ['$'] and buffer == ['$']:
      print("{:>20} {:>20} {:>20}"
         .format(' '.join(buffer),
               ' '.join(stack),
               "Valid"))
      return "\nValid String!"
   elif stack[0] not in term_userdef:
      # take font of buffer (y) and tos (x)
      x = list(diction.keys()).index(stack[0])
      y = table_term_list.index(buffer[-1])
      if parsing_table[x][y] != '':
```

```
            # format table entry received
            entry = parsing_table[x][y]
            print("{:>20} {:>20} {:>25}".
               format(' '.join(buffer),
                      ' '.join(stack),
                      f"T[{stack[0]}][{buffer[-1]}] = {entry}"))
            lhs_rhs = entry.split("->")
            lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
            entryrhs = lhs_rhs[1].split()
            stack = entryrhs + stack[1:]
         else:
            return f"\nInvalid String! No rule at " \
               f"Table[{stack[0]}][{buffer[-1]}]."
      else:
         # stack top is Terminal
         if stack[0] == buffer[-1]:
            print("{:>20} {:>20} {:>20}"
               .format(' '.join(buffer),
                      ' '.join(stack),
                      f"Matched:{stack[0]}"))
            buffer = buffer[:-1]
            stack = stack[1:]
         else:
            return "\nInvalid String! " \
               "Unmatched terminal symbols"



sample_input_string = None

# sample set 1 (Result: Not LL(1))
# rules=["A -> S B | B",
#    "S -> a | B c | #",
#    "B -> b | d"]
# nonterm_userdef=['A','S','B']
# term_userdef=['a','c','b','d']
# sample_input_string="b c b"

# sample set 2 (Result: LL(1))
# rules=["S -> A | B C",
#    "A -> a | b",
#    "B -> p | #",
#    "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="p c"

# sample set 3 (Result: LL(1))
# rules=["S -> A B | C",
```

```
#    "A -> a | b | #",
#    "B-> p | #",
#    "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="a c b"


# sample set 4 (Result: Not LL(1))
# rules = ["S -> A B C | C",
#       "A -> a | b B | #",
#       "B -> p | #",
#       "C -> c"]
# nonterm_userdef=['A','S','B','C']
# term_userdef=['a','c','b','p']
# sample_input_string="b p p c"


# sample set 5 (With left recursion)
# rules=["A -> B C c | g D B",
#    "B -> b C D E | #",
#    "C -> D a B | c a",
#    "D -> # | d D",
#    "E -> E a f | c"
#    ]
# nonterm_userdef=['A','B','C','D','E']
# term_userdef=["a","b","c","d","f","g"]
# sample_input_string="b a c a c"


# sample set 6
# rules=["E -> T E'",
#    "E' -> + T E' | #",
#    "T -> F T'",
#    "T' -> * F T' | #",
#    "F -> ( E ) | id"
# ]
# nonterm_userdef=['E','E\'','F','T','T\'']
# term_userdef=['id','+','*','(',')']
# sample_input_string="id * * id"
# example string 1
# sample_input_string="( id * id )"
# example string 2
# sample_input_string="( id ) * id + id"
```

```
# sample set 7 (left factoring & recursion present)
rules=["S -> A k O",
   "A -> A d | a B | a C",
   "C -> c",
   "B -> b B C | r"]


nonterm_userdef=['A','B','C']
term_userdef=['k','O','d','a','c','b','r']
sample_input_string="a r k O"


diction = {}
firsts = {}
follows = {}


# computes all FIRSTs for all non terminals
computeAllFirsts()
# assuming first rule has start_symbol
# start symbol can be modified in below line of code
start_symbol = list(diction.keys())[0]
# computes all FOLLOWs for all occurrences
computeAllFollows()


(parsing_table, result, tabTerm) = createParseTable()

# validate string input using stack-buffer concept
if sample_input_string != None:
   validity = validateStringUsingStackBuffer(parsing_table, result,
                                  tabTerm, sample_input_string,
                                  term_userdef,start_symbol)
   print(validity)
else:
   print("\nNo input String detected")
```

Output:

Rules:

S->[['A', 'k', 'O']]
A->[['A', 'd'], ['a', 'B'], ['a', 'C']]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]

After elimination of left recursion:

S->[['A', 'k', 'O']]
A->[['a', 'B', "A'"], ['a', 'C', "A'"]] C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]

After left factoring:

S->[['A', 'k', 'O']]
A->[['a', "A'"]]
A''->[['B', "A'"], ['C', "A'"]]
C->[['c']]
B->[['b', 'B', 'C'], ['r']]
A'->[['d', "A'"], ['#']]

Calculated firsts:
first(S) => {'a'}
first(A) => {'a'}
first(A'') => {'r', 'c', 'b'}
first(C) => {'c'}
first(B) => {'r', 'b'}
first(A') => {'d', '#'}

Calculated follows:
follow(S) => {'$'}
follow(A) => {'k'}
follow(A'') => {'k'}
follow(C) => {'c', 'k', 'd'}

follow(B) => {'c', 'k', 'd'}
follow(A') => {'k'}

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
S	{'a'}	{'$'}
A	{'a'}	{'k'}
A''	{'r', 'c', 'b'}	{'k'}
C	{'c'}	{'c', 'k', 'd'}
B	{'r', 'b'}	{'c', 'k', 'd'}
A'	{'d', '#'}	{'k'}

Generated parsing table:

	k	O	d	a	c	b	r	$
S				S->A k O				
A				A->a A''				
A''					A''->C A'	A''->B A'	A''->B A'	
C					C->c			
B						B->b B C	B->r	
A'	A'->#		A'->d A'					

Validate String => a r k O

Buffer	Stack	Action
$ O k r a	S $	T[S][a] = S->A k O
$ O k r a	A k O $	T[A][a] = A->a A''
$ O k r a	a A'' k O $	Matched:a
$ O k r	A'' k O $	T[A''][r] = A''->B A'
$ O k r	B A' k O $	T[B][r] = B->r
$ O k r	r A' k O $	Matched:r
$ O k	A' k O $	T[A'][k] = A'->#
$ O k	k O $	Matched:k
$ O	O $	Matched:O
$	$	Valid

Valid String!

SPCC
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
Arnav Malvia
C23
2103109

Experiment:- 5

Aim:- Write a program to implement Three Address code (TAC).

Theory:-

In Three address code each statement generally contains three addresses, two for operands and for the result. This is why it it termed as three address code. It is a popular form of intermediate code used in optimising compilers. It is a sequence of statements having a general form.

$$a = b \ op \ c$$

Where op is any operator like +, -, *, etc and a, b and c can be variables, constants or the temporary variables generated by the compiler. We can say that three address code is linearised representation of a syntax tree. In general when generating three address statements, the compiler has to create new temporary variables as needed. The advantage of using temporary names for intermediate results is that the TAC can be easily rearranged. We use a function newtemp() that returns a new temporary each time it is called. Temporary names must be generated to compute intermediate operations. Addresses are implemented as pointers to their symbol table entries.

Implementation of Three Address statement:-

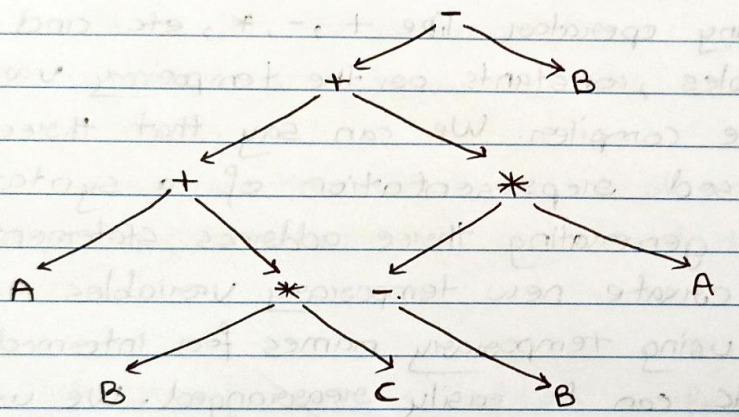
A three address code can be implemented as records having fields for the operations and operands. TAC is a form of representing intermediate code used by compilers to aid in the implementation of code improving transformations. In a compiler it can be realised with

the fields for the operators and the operands. The TAC can be implemented in several ways, they are as:

i) Quadruples
ii) Triples
iii) Indirect Triples

Example:- Construct Syntax tree and three address code for the following string (A+B*C)+(-B*A)-B

Syntax tree:-



Three Address Code:-

$t_1 = B * C$

$t_2 = A + t_1$

$t_3 = -B$

$t_4 = t_3 * A$

$t_5 = t_2 + t_4$

$t_6 = t_5 - B$

19/3/24 A+

Conclusion:- Understood and Implemented Three Address Code.

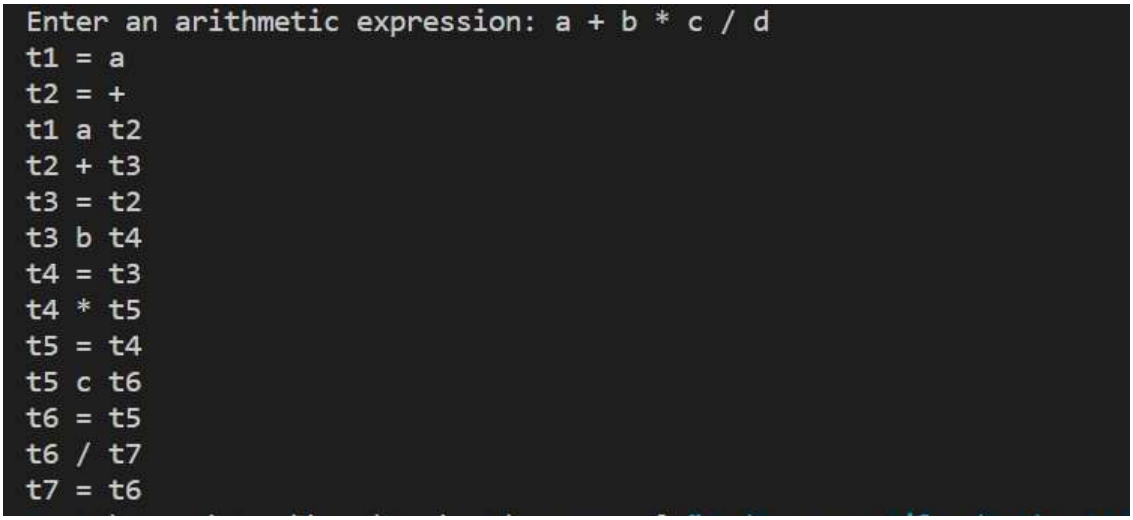
Experiment 5 :-

Code:

```
def generate_tac(expr):
    # Split the input expression into individual tokens
tokens = expr.split()

    # Initialize variables for the three-address
code     t_count = 1     t_prefix = "t"     code =
[]

    # Generate the three-address code
    for i in range(len(tokens)):
        if i == 0:
            code.append(f"{t_prefix}{t_count} = {tokens[i]}")
        elif i == 1:
            t_count += 1             code.append(f"{t_prefix}{t_count} =
{tokens[i]}")             code.append(f"{t_prefix}{t_count-1}  {tokens[i-
1]}  {t_prefix}{t_count}")
        else:
            t_count += 1
            code.append(f"{t_prefix}{t_count-1}  {tokens[i-1]}
{t_prefix}{t_count}")             code.append(f"{t_prefix}{t_count} =
{t_prefix}{t_count-1}")

    # Print the three-address code
    for line in code:
print(line)
```

```
# Prompt the user for an input expression
expr = input("Enter an arithmetic expression: ")

# Generate and print the corresponding three-address code
generate_tac(expr)
```

OUTPUT:



Arnav Malvia
SPCC
C23
2103109

SPCC
Arnav
Malvia
C23
2103109

Experiment :- 6

Aim :- Write a program to implement Code Optimisation.

Theory :-

The code optimisation in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources. so that faster running machine code will result.

Compiler optimising process should meet the following objectives:

1) The optimisation must be correct, it must not in any way change the meaning of the program.
2) Optimisation should increase the speed and performance of the program.
3) The compilation time must be kept reasonable.
4) The optimisation process should not delay the overall compiling process.

Optimisation of code is often performed at the end of development stage since it reduces readability and adds code that is used to increase the performance.

Optimisation helps to :-

1) Reduce the space consumed and increases the speed of compilation.
2) Manually analyzing datasets involves lots of time. Hence we make use of software like Tableau for analysis. Similarly manually performing the optimisation is also tedious and is better done using a code optimizer.
3) An optimsed code often promotes re-usability.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Types of code optimisation:-

1) Machine Independent Optimisation:
This code optimisation phase attempts to improve the intermediate code to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

2) Machine Dependent Optimisation:
Machine dependent optimisation is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine dependent optimisers put efforts to take maximum advantage of the memory hierarchy.

Conclusion:- Understood and implemented code optimisation.

19/3/24

A+

Experiment 6 :-

Code:

```
//Code Optimization Technique
#include<stdio.h>
#include<string.h>
struct op
{
char l;
char r[20];
}
op[10],pr[10];
void main()
{
int a,i,k,j,n,z=0,m,q;
char *p,*l;
char temp,t;
char *tem;
printf("Enter the 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("Intermediate Code\n") ;
for(i=0;i<n;i++)
{
```

```
printf("%c=",op[i].l);
printf("%s\n",op[i].r);
}
for(i=0;i<n-1;i++)
{
temp=op[i].l;
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++;
}
}
}
pr[z].l=op[n-1].l;
strcpy(pr[z].r,op[n-1].r);
z++;
printf("\nAfter Dead Code Elimination\n");
for(k=0;k<z;k++)
{
printf("%c\t=",pr[k].l);
printf("%s\n",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-pr[i].r;
printf("pos: %d\n",a);
pr[i].r[a]=pr[m].l;
}}}}}
printf("Eliminate Common Expression\n");
for(i=0;i<z;i++)
{
printf("%c\t=",pr[i].l);
printf("%s\n",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)
```

```
{
pr[i].l='\0';
}
}
}
printf("Optimized Code\n");
for(i=0;i<z;i++)
{
if(pr[i].l!='\0')
{
printf("%c=",pr[i].l);
printf("%s\n",pr[i].r);
}
}
}
```

Output:

```
Enter the Number 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
```

```
pos: 2
Eliminate Common Expression
b   =c+d
b   =c+d
f   =b+b
r   =f
Optimized Code
b=c+d
f=b+b
r=f
```

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
Arnav Malvia
C23
2103109

Experiment :- 7

Aim :- Write a program to implement Pass1 and Pass 2 of Multi-Pass Assembler.

Theory :-

PASS 1:

1) Input Processing :-
In pass 1, the assembler reads the entire assembly language program line by line.

2) Symbol Table Creation :-
As assembler encounters lables, it adds them to a symbol table along with their corresponding memory address.

3) Location Counter (LC) Management :-
The assembler maintains a location counter (LC) that keeps track of memory address addressed to instructions and data.

4) Error Detection :-
Pass 1 also performs basic error checks such as syntax errors, undefined symbols and duplicate labels

5) Output :-
Pass 1 generates an intermediate file containing the symbol table and modified assembly code with resolved address

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Pass 2:

1) I/p Processing-
Pass 2 takes the intermediate file generated by PASS 1 as i/p, it reads the modified assembly code and symbols.

2) Instruction Translation-
For each instruction Pass 2 translates the mnemonic opcode and operand(s) into corresponding machine code.

3) Object Code Generation-
As pass 2 processes each instruction, it generates the object code in binary or hexadecimal format.

4) ~~Final o/p~~-
~~Upon successful comple~~

4) Error Handling-
Pass 2 performs additional error checks such as undefined symbols (not resolved in PASS 1), invalid opcodes and operand mismatches.

5) Final o/p-
Upon successful completion, Pass 2 produces the final machine code or ~~to~~ object program ready for execution on target computer.

Conclusion:-
The implementation of Pass 1 and Pass 2 in a multi-pass assembler involves efficient i/p processing, address resolution, error detection, and obj code generation.

26/3/24 A+

Experiment 7 :-

Code:

Source_Code.asm

```
START    LDA     VAL1      ; Load accumulator with value
         ADD     VAL2      ; Add value 2 to accumulator
         STP     RESULT    ; Store the result in memory
         HLT               ; Halt the program
VAL1     DAT       1       ; Data: value 1
VAL2     DAT       2       ; Data: value 2
RESULT   DAT       0       ; Data: Result
```

firstpass.py

```
def first_pass_assembler(source_code):
    symbol_table = {}
    location_counter = 0

    with open(source_code, 'r') as file:
        for line in file:
            line = line.split(';')[0].strip()
            if not line:
                continue

            tokens = line.split()
            label = None
            if len(tokens) > 1:
                label = tokens[0]
```

```
            if label and label in symbol_table:
                print(f"Error: Duplicate label '{label}'")
                return None

            if label:
                symbol_table[label] = location_counter

            location_counter += 1

    return symbol_table

# Example usage:
source_code = "Source_Code.asm"
symbol_table = first_pass_assembler(source_code)
if symbol_table:
    print("Symbol Table:")
    for label, location in symbol_table.items():
        print(f"{label}: {location}")
```

Output:

```
PS C:\Users\91992\Downloads>  & 'c:\Users\91992\AppData\Local\Programs\Python\Python312\python.exe'
ugpy\adapter/../..\debugpy\launcher' '49366' '--' 'c:\Users\91992\Downloads\firstpass.py'
Symbol Table:
START: 0
ADD: 1
STP: 2
VAL1: 4
VAL2: 5
RESULT: 6
```

secondpass.py

```
def pass2_assembler(source_code, symbol_table):
    machine_code = []

    with open(source_code, 'r') as file:
        for line in file:
            line = line.split(';')[0].strip()
            if not line:
                continue

            tokens = line.split()
            translated_instruction = ''

            for i, token in enumerate(tokens):
                if token.isdigit():
                    translated_instruction += token + ' '
                elif token in symbol_table:
                    translated_instruction += str(symbol_table[token]) + ' '
                elif token == 'DAT':
                    translated_instruction += tokens[i+1] + ' '
                else:
                    translated_instruction += token + ' '

            machine_code.append(translated_instruction.strip())
```

```
    return machine_code


# Example usage:
source_code = "Source_Code.asm"
symbol_table = {
    'START': 0,
    'ADD': 1,
    'STP': 2,
    'VAL1': 4,
    'VAL2': 5,
    'RESULT': 6
}
machine_code = pass2_assembler(source_code, symbol_table)
print("Machine Code:")
for instruction in machine_code:
    print(instruction)
```

Output:

```
PS C:\Users\91992\Downloads>  c:; cd 'c:\Users\91992\Downloads'; & 'c:\Users\91992\AppData\Loca
2024.2.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '49382' '--' 'c:\Users'
Machine Code:
0 LDA 4
1 5
2 6
HLT
4 1 1
5 2 2
6 0 0
PS C:\Users\91992\Downloads>
```

```
PS C:\Users\91992\Downloads>  & 'c:\Users\91992\AppDa
ugpy\adapter/../..\debugpy\launcher' '49366' '--' 'c:
Symbol Table:
START: 0
ADD: 1
STP: 2
VAL1: 4
VAL2: 5
RESULT: 6
PS C:\Users\91992\Downloads>  c:; cd 'c:\Users\91992\
2024.2.0-win32-x64\bundled\libs\debugpy\adapter/../..
Machine Code:
0 LDA 4
1 5
2 6
HLT
4 1 1
5 2 2
6 0 0
PS C:\Users\91992\Downloads>
```

Arnav
Malvia
C23
2103109
SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment :- 8

Aim:- Write a program to implement Multi Pass Macroprocessor.

Theory:-

A multi pass macroprocessor is a type of macroprocessor that processes macros in multiple passes over the source code.

1) Macro Expansion-

In the first pass the macro definitions are scanned and processed to expand macros wherever they are invoked in the source code. Macros are typically expanded by replacing their names with their corresponding definitions.

2) Text Substitution-

The expanded code resulting from macro expansion may contain other macros or conditional statements. The multi-pass macroprocessor iterates over the expanded code, performing text substitution for any remaining macros or processing conditional directives.

3) Iteration-

Depending on the complexity of the macros and the presence of conditional statements, the macroprocessor may need to iterate over the source code multiple times to ensure all macros are fully expanded.

4) Conditional Compilation-

Multi pass macroprocessors may also handle conditional

compilation directives such as #ifdef, #ifndef, #if, #elif and #endif.

5) Error Handling-
Multipass macroprocessors typically perform error checking and reporting during each pass to detect and report any syntax errors.

6) Optimisation-
Some multi pass microprocessors may incorporate optimization techniques to improve the efficiency of macro expansion and text substitution.

7) Output Generation-
After all passes are completed and ~~it~~ the source code is fully processed, the multi pass macroprocessor generates the final output which may be assembly code, object code or another form of executable code depending on the target platform and compiler.

Conclusion:- Understood and implemented Multi Pass Macroprocessor.

26/3/24
A

Experiment 8 :-

CODE:

```
# First Pass - Identifying and storing macros macro_table = {}
# Dictionary to store macro definitions with open('input.txt',
'r') as file:
    line_count = 0     for line in file:
line_count += 1         tokens =
line.strip().split(' ')         if
tokens[0] == 'MACRO':
macro_name = tokens[1]
parameters = tokens[2:]
macro_definition = []             line =
file.readline().strip()             while
line != 'MEND':
                macro_definition.append(line)
line = file.readline().strip()
            macro_table[macro_name] = (parameters, macro_definition)


# Second Pass - Expanding macros
output = [] with open('input.txt', 'r')
as file:
    line_count = 0     for line in file:
line_count += 1         tokens =
line.strip().split(' ')         if
tokens[0] == 'MACRO':

            continue    # Skip macro definition lines
elif tokens[0] in macro_table:             # Expand
macro definition
```

```
        parameters, macro_definition = macro_table[tokens[0]]
parameter_values = tokens[1:]            if len(parameter_values) !=
len(parameters):
            raise Exception(f'Error: Number of parameters does not match at line
{line_count}')            for i in
range(len(parameters)):
            macro_definition = [x.replace(parameters[i], parameter_values[i]) for x in
macro_definition]
        output.extend(macro_definition)
else:
        # Non-macro line, add to output
output.append(line.strip())

# Print output and data structures used
print('Output:') for line in output:
   print(line)

print('Macro Table:') for
macro_name in macro_table:
   parameters, macro_definition = macro_table[macro_name]
print(f'{macro_name}: {parameters} {macro_definition}')
```

INPUT FILE:

```
MACRO ADD NUM1, NUM2
MOV AX, NUM1
ADD AX, NUM2
MOV RESULT, AX
MEND

MACRO SUBTRACT NUM1, NUM2
MOV AX, NUM1
```

SUB AX, NUM2

MOV RESULT, AX

MEND

START

ADD 10, 20

SUBTRACT 50, 30

END

OUTPUT:

```
Output:
MOV AX, NUM1
MOV AX, NUM1
ADD AX, NUM2
MOV RESULT, AX
MOV RESULT, AX
MEND

MOV AX, NUM1
SUB AX, NUM2
MOV RESULT, AX
MEND

START
MOV AX, NUM1
ADD AX, 20
MOV RESULT, AX
MOV AX, NUM1
SUB AX, 30
MOV RESULT, AX
END
Macro Table:
ADD: ['NUM1,', 'NUM2'] ['MOV AX, NUM1', 'ADD AX, NUM2', 'MOV RESULT, AX']
SUBTRACT: ['NUM1,', 'NUM2'] ['MOV AX, NUM1', 'SUB AX, NUM2', 'MOV RESULT, AX']
PS C:\Users\Harsh\Desktop\python new>
```

Asmay Malvia
C23
2103109
SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment :- 9

Aim :- Write a program to implement Code Generation.

Theory :-
Code generation can be considered as the final phase of compilation, through post code generation optimisation can be applied on the code but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some low level programming language.
eg :- assembly language.

Code generator is used to produce the target code for three address statements. It uses registers to store the operands of three address statements.

Register and address descriptions:
A register description contains the track of what is currently in each register. The register description shows that all registers are initially empty.

An address descriptor is used to store the location where current value of name can be found at run time.

Algorithm :-
The algorithm takes a sequence of three address statements as input. For each three address statements of the form a=b op c perform the various actions as follows-

1) Invade a function getting to find out the location L where the result of computation b op c should be stored.

2) Consult the address description for y to determine y. If the value of y currently in memory and register both then prefer the register 'y'. If value of y is not already in L then generate the instruction mov y, L to place a copy of y in L.

3) Generate the instruction OP z', L where z' is used to show the current location of z. If z is in both then prefer a register to a memory location. If x is in L then update.

4) If the current value of y or z have no next uses or not live on exit, From the block or in register then after the register descriptor to indicate those registers will no longer contain y or z.

Conclusion:- Understood and implemented code generation.

26/3/24

A

Experiment 9 :-

Code:

```
def removeLeftRecursion(rulesDiction):
    store = {}
    for lhs in rulesDiction:
        alphaRules = []
        betaRules = []
        allrhs = rulesDiction[lhs]
        for subrhs in allrhs:
            if subrhs[0] == lhs:
                alphaRules.append(subrhs[1:])
            else:
                betaRules.append(subrhs)
        if len(alphaRules) != 0:
            lhs_ = lhs + "'"
            while (lhs_ in rulesDiction.keys()) or (lhs_ in store.keys()):
                lhs_ += "'"
            for b in range(0, len(betaRules)):
                betaRules[b].append(lhs_)
            rulesDiction[lhs] = betaRules
            for a in range(0, len(alphaRules)):
                alphaRules[a].append(lhs_)
            alphaRules.append(['#'])
            store[lhs_] = alphaRules
    for left in store:
        rulesDiction[left] = store[left]
    return rulesDiction

# Example grammar rules
rulesDiction = {
    'A': [['A', 'a'], ['A', 'b'], ['c']],
    'B': [['B', 'x'], ['y']]
}

# Apply left recursion elimination
result = removeLeftRecursion(rulesDiction)

# Print the modified rules
for key, value in result.items():
    print(f"{key} -> {value}")
```

Output:

Aman Malviya C23 2103109 5PCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment:- 10

Aim:- Write a program to eliminate left recursion from the given grammar.

Theory:-

A grammar G (V,T,P,S) is left recursive if it has a production in the form, $A \rightarrow A\alpha|\beta$.

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

$A \rightarrow BA'$

$A \rightarrow \alpha A' | \varepsilon$

Elimination of left recursion: Left recursion can be eliminated by introducing new non terminal A such that

$A \rightarrow A\alpha|\beta$ — Removal of left recursion → $A \rightarrow BA'$, $A' \rightarrow \alpha A'|\varepsilon$

Left Recursive Grammar.

The This type of recursion is also called immediate left recursion. In left recursive grammar expansion of A will generate $A\alpha$, $A\alpha\alpha$, $A\alpha\alpha\alpha$ at each step causing it to enter into an infinite loop.

The general form for left recursion is

$A \rightarrow A\alpha_1 | A\alpha_2 \ldots A\alpha_m | \beta_1 | \beta_2 \ldots | \beta_n$ can be replaced by

$A \rightarrow \beta_1 | \beta_1 A' | \beta_2 A' | \ldots \beta_n A'$

$A \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$

eg:- $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Eliminate left recursion from the grammar

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

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

$A \rightarrow \beta A'$ means $E \rightarrow TE'$
$A' \rightarrow \alpha A' \mid \varepsilon$ means $E' \rightarrow +TE' \mid \varepsilon$

Comparing $T \rightarrow T*F \mid F$ with $A \rightarrow A\alpha \mid \beta$
$T \rightarrow T*F \mid F$
$A \rightarrow A\alpha \mid \beta$
$A : T \quad \alpha : *F \quad \beta = F$

$A \rightarrow \alpha A' \mid \varepsilon$ means $T' \rightarrow *FT' \mid \varepsilon$
$A \rightarrow \beta A'$ means $T \rightarrow FT'$
Production $F \rightarrow (E) \mid id$ does not have any left recursion.

Combining all solutions we have:
$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

(A)

26/3/24

Conclusion:- Understood and implemented elimination of left recursion.

Experiment 10 :-

CODE:

```
def main():
  n = int(input("Enter the number of expressions: "))

  expressions = {}
  expr_registers = {}
  codeGen = []
  count = 0

  opcodes = {
    "+": "ADD",
    "-": "SUB",
    "*": "MUL",
    "/": "DIV"
  }

  for i in range(n):
    exp = input(f"Enter TAC expression {i + 1} : ")

    lhs, rhs = exp.split('=')
    expressions[lhs] = rhs

    op1 = rhs[0]
    op = rhs[1]
    op2 = rhs[2]

    expr_registers[lhs] = f"R{count}"

    if expr_registers.get(op1) is None or expr_registers.get(op2) is None:
      code = f"MOV {op1}, R{count} \n{opcodes[op]} {op2}, R{count}"
      codeGen.append(code)
      count += 1
    else:
      expr_registers[lhs] = expr_registers[op1]
      code = f"{opcodes[op]} {expr_registers[op2]}, {expr_registers[op1]}"
```

```
            codeGen.append(code)


    # print(expr_registers)

    print("--"*10)
    print("Generated Code is: ")
    print("--" * 10)
    for code in codeGen:
        print(code)
        print()


if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter the number of expressions: 4
Enter TAC expression 1 : t=a-b
Enter TAC expression 2 : u=a-c
Enter TAC expression 3 : v=t+u
Enter TAC expression 4 : d=v+u
--------------------
Generated Code is:
--------------------
MOV a, R0
SUB b, R0

MOV a, R1
SUB c, R1

ADD R1, R0

ADD R1, R0
```

Process finished with exit code 0

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
SPCC
Arnav Malvia
C23
2103109

Assignment :- 1

Q1 With reference to assembler, consider any assembly program and generate Pass-1 and Pass-2 output. Also show contents of Database table involved in it.

Ans1 Assembly language is a low-level programming language that is specific to a particular computer architecture or processor.

Assembly language assembler is a software tool that translates assembly language code into machine code, which can be executed by a machine's CPU. It performs this translation process in two main phases phase 1 and phase 2.

Two pass assembly process:-

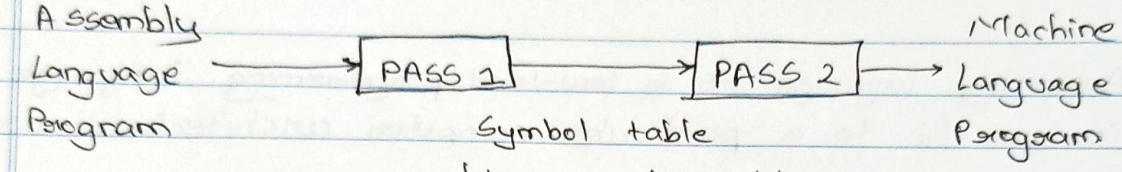
Two pass assembly process is a common method used by assembler to translate assembly language source code into machine code. It involves two main processes over the source code.

Pass-1 :- Scans the code, validates the tokens, creates symbol table.

i) It keeps track of location counter (LC)
ii) Determines the length of machine instructions (MOT).
iii) Keeps track of values of symbols until pass-2 is done (ST).
iv) Process some pseudo-oprode eg: EQU, DS DC (POT)
v) Stores the Literals (LT)

Pass-2 :- Converts the code to machine code, solves forward reference problem.

i) It looks up the values of symbols (ST).
ii) Generates the instructions (MOT).
iii) Generates the data (For DS, DC and literals)
iv) Processes some pseudo-opcodes eg: USING, DROP (POT)



Sample program:-

```
PRG      START   0
         MACRO
&LAB     INCR    &ARG1,&ARG2,&ARG3
&LAB       A     1,&ARG1
           A     2,&ARG2
           A     3,&ARG3
         MEND
LOOP1    INCR    DATA1,DATA2,DATA3
DATA1      DC    F'5'
DATA2      DC    F'10'
DATA3      DC    F'15'
```

Pass 1 output:

```
PRG      START   0
LOOP1    INCR    DATA1,DATA2,DATA3
DATA1      DC    F'5'
DATA2      DC    F'10'
DATA3      DC    F'15'
```

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Pass 1 Data Structures used:-

1. Macro Definition Table (MDT)

Index	Name		
1	&LAB	INCR	&ARG1, &ARG2, &ARG3
2	#0	A	1, #1
3		A	2, #2
4		A	3, #3
5		MEND	

2. Macro Name Table (MNT)

Index	Name	MDT Index
1	INCR	1

3. Argument List Array (ALA)

Index	Arguments
0	
1	
2	
3	

Pass 2 output:

PRG	START	0
LOOP 1	A	1, DATA 1
	A	2, DATA 2
	A	3, DATA 3

```
DATA 1      DC      F'5'
DATA 2      DC      F'10'
DATA 3      DC      F'15'
```

1) Argument ~~Array~~ List Array (ALA)

Index	Arguments
0	LOOP1
1	DATA 1
2	DATA 2
3	DATA 3

Q2 Write a short note on YACC.

Ans 2 YACC or Yet Another Compiler-Compiler is a powerful tool used in the field of compiler construction to automate the process of generating parser for programming language. It plays control role in transforming high level language specification, typically written in formal grammar notation such as BNF into efficient parser capable of recognizing and analyzing the structure of input program.

YACC functionality and how it fits into the compiler construction process:-

1) Formal Grammar specification:
YACC token input a formal specification of the syntax rules for a programming language. This specification defines the language syntax in terms of production rules which describe how valid sentence or program can be constructed from the language symbol.

2) Lexer Integration:
YACC works in conjugation with lexical tool such as lex or

Flex. Lexer are responsible for breaking down the input text into token or lexemes, which are the smallest meaningful unit in the language.

3) Parser generation:
YACC generates a parser for the specified grammar based on the principles of LR parsing. LR parsing is a bottom up parsing technique that builds a parse tree for input string starting from the leaves and working up to the root.

4) Parsing Algorithm:
YACC parser implements an efficient parsing algorithm that navigates through grammar rules and inputs token whenever input confirms to the language syntax.

5) Error handling:
YACC parser can be augmented with error handling mechanism to detect and report syntax error in the input. Error recovery strategies can also be determined to help the parse continues parsing after encountering error, ensuring that multiple syntax error in program do not halt the parsing process prematurely.

19/3/24

A+

Arnav Malvia
C23
2103109
SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Assignment :- 2

Q1 Explain the working of Direct Linking Loader with example showing the entries in different databases built in DLL.

Ans 1 A direct linking loader is a component of an operating system responsible for loading and executing a program directly from memory. The loader is typically used in systems where the entire program is loaded into memory before execution, and the memory addresses of all external references are known at compile time.

Working of direct linking loader:-

1. Loading into memory:
The loader reads the object file or executable binary into memory. This could involve fetching the program from a disk or any other storage device into RAM.

2. Symbol Resolution:
In direct linking, all external symbols (functions or variables) are resolved at compile time. This means that during compilation, the compiler knows the address of all symbols, referenced in the program.

3. Binding Addresses:
The loader binds the addresses of external symbols directly into the program's code or data sections. This binding involves replacing placeholder addresses with the actual addresses are known at compile time, the loader can directly embed them into the program.

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

4. Execution:

Once all symbols are bound, the program is ready for execution. Control is transferred to the entry point of the program, and it begins execution.

Consider an example involving a DLL named 'database dll' that contains functions to interact with different databases.

```
// database.dll
// function to retrieve data from MySQL database
int RetrieveFromMySQL(char *query);

// function to store data in MongoDB database
void StoreInMongoDB(char* data);
```

~~## function~~

Additionally suppose we have an application program 'main-exe' that uses functions from database.dll.

```
//main.exe.

#include <stdio.h>
#include< database.h>

int main() {
  char * data = "sample data";

// call function to retrieve data from MySQL.
```

```
int result = RetrieveFromMySQL("SELECT * FROM table");
printf("Retrieved %d records\n", result);

// Call function to store data in MongoDB
StoreInMongoDB(data);

return 0;
}
```

Execution flow:-

1) Compilation of main.exe: During compilation of main.exe, the compiler links against the symbols declared in ~~the compiler~~ database.dll. The compiler knows the addresses of the 'RetrieveFromMySQL' and 'StoreInMongoDB' because they are declared in 'Database.h' header file.

2) Symbol binding: The addresses of 'RetrieveFromMySQL' and 'StoreInMongoDB' are directly embedded into the code sections of 'main.exe' during compilation.

3) Loading and Execution: When 'main.exe' is loaded by the operating system, the direct linking loader loads the program into memory. It binds the addresses, allowing the program to call those functions directly without needing to perform any further symbol resolution at runtime.

26/3/24

Hence the above example involving entries ~~can~~ in different databases built into a DLL demonstrates how direct linking works in practice.

A+

Aarnav Malvia
C23
2103109
SPCC

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Assignment :-3

Q1 Write Short Note on :-
a) Editors and type of editors
b) Backpatching in Intermediate Code Generation

Ans 1

a) Editors and type of editors:-

Editors or text editors are software programs that enable users to create and edit text files. In the field of programming, the term editor usually refers to source code editors that include many features for writing and editing code.

Types of editors:-

1) Line editors - In this type you can edit only one line at a time or an integral number of lines.

2) Stream Editors - In this type the file is treated as continuous slow or sequence of characters instead of line members, which means here you can type paragraphs.

3) Screen Editors - In this type the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily.

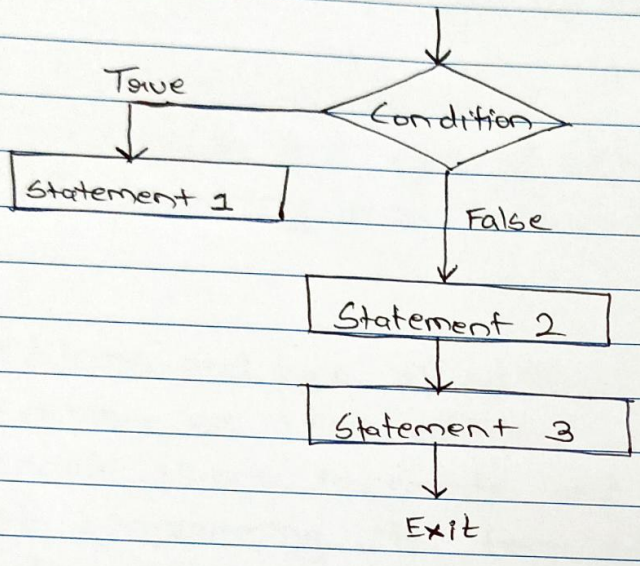
4) Word Processors - Overcoming the limitations of screen editors it allows one to use some format to insert images, videos, style, font, size, etc.

5) Structure Editor - Structure editor focusses on programming languages. It provides features to write and edit source code.

b) Backpatching in Intermediate Code Generation :-
Backpatching is basically a process of levelling unspecified information. This information is ~~also~~ of labels. It basically uses the appropriate semantic actions during the process of code generation. It may indicate the address of the label in goto statements while producing TAC's bar the given expressions. ~~Hence~~ Here basically two passes are used because aligning the positions of these labels statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in the second round.

Need for backpatching :-
i) Boolean expression - Statements ~~write~~ where results are either true or false. A boolean expression which is named after mathematician George Boole, is an expression that evaluates to either True or False.

ii) How to control statements - The blow of control statements needs to be controlled during the execution of statements in a program.



A+

26/3/24