

**Department: Computer Science and Engineering**

**Academic Year : 2025 - 2026**

**Semester : V**

**Staff Name : Dr.I.Vallirathi**

**Designation : Associate Professor**

**Department : Computer Science and Engineering**

**Regulation :2021**

**LAB MANUAL**

**CS 3501 – COMPILER DESIGN LAB COMPONENT**

|  |  |
| --- | --- |
| **PREPARED BY** | **VERIFIED BY** |
| **Dr.I.Vallirathi** | **HOD** |

**PROGRAM OUTCOMES (POs)**

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

**CS3501 COMPILER DESIGN L T P C 3 0 2 4**

**LIST OF EXPERIMENTS:**

1. Using the LEX tool, Develop a lexical analyzer to recognize a few patterns in C. (Ex.

identifiers, constants, comments, operators etc.). Create a symbol table, while recognizing

identifiers.

2. Implement a Lexical Analyzer using LEX Tool

3. Generate YACC specification for a few syntactic categories.

a. Program to recognize a valid arithmetic expression that uses operator +, -, \* and /.

b. Program to recognize a valid variable which starts with a letter followed by any number of letters

or digits.

c. Program to recognize a valid control structures syntax of C language (For loop, while loop, if-

else, if-else-if, switch-case, etc.).

d. Implementation of calculator using LEX and YACC

4. Generate three address code for a simple program using LEX and YACC.

5. Implement type checking using Lex and Yacc.

6. Implement simple code optimization techniques (Constant folding, Strength reduction and Algebraic transformation)

7. Implement back-end of the compiler for which the three address code is given as input and the 8086

assembly language code is produced as output.

**30 PERIODS**

**COURSE OUTCOMES:**

On Completion of the course, the students should be able to:

**CO1:**Understand the techniques in different phases of a compiler.

**CO2:**Design a lexical analyser for a sample language and learn to use the LEX tool.

**CO3:**Apply different parsing algorithms to develop a parser and learn to use YACC tool

**CO4:**Understand semantics rules (SDT), intermediate code generation and run-time environment.

**CO5:**Implement code generation and apply code optimization techniques.

|  |  |  |
| --- | --- | --- |
| **Ex:No** | **LIST OF EXPERIMENTS** | **No of Hours Planned** |
| 1 | Develop a Lexical Analyzer to recognize a few patterns in C. | 3 |
| 2 | Implementation of Lexical Analyzer using Lex Tool | 3 |
| 3 | 1. Program to recognize a valid arithmetic expression that uses operator +, - , \* and / | 3 |
|  | 1. Program to recognize a valid variable which starts with a letter followed by any number of letters or digits | 3 |
|  | 1. Implementation of calculator using Lex and Yacc | 4 |
| 4 | Generate three address code using Lex and Yacc | 5 |
| 5 | Implementation of Type Checking | 3 |
| 6 | Implementation of simple code optimization techniques | 3 |
| 7 | Implementation of back end of a compiler | 3 |
| **TOTAL PLANNED** | | **30** |

**Ex. No. 1 DEVELOP** **A** **LEXICAL** **ANALYZER** **TO** **RECOGNIZE** **A** **FEW** **PATTERNS** **IN** **C.**

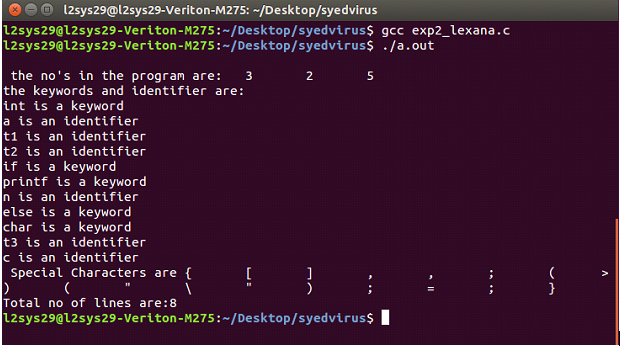
**(EX. IDENTIFIERS,** **CONSTANTS,** **COMMENTS,** **OPERATORS** **ETC.).**

**AIM:**  
To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using C program  
  
**ALGORITHM:**  
**Step1:** Start the program.  
**Step2:** Declare all the variables and file pointers.  
 **Step3:** Display the input program.  
**Step4:** Separate the keyword in the program and display it.  
**Step5:** Display the header files of the input program  
**Step6:** Separate the operators of the input program and display it.  
 **Step7:** Print the punctuation marks.  
**Step8:**Print the constant that are present in input program.  
**Step9:** Print the identifiers of the input program.

**PROGRAM CODE:**

//Develop a lexical analyzer to recognize a few patterns in C.

#include<string.h>  
#include<ctype.h>  
#include<stdio.h>  
#include<stdlib.h>  
void keyword(char str[10])  
{  
 if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||strcmp("double",str)==0||strcmp("printf",str)==0||strcmp("switch",str)==0||strcmp("case",str)==0)  
  printf("\n%s is a keyword",str);  
 else  
  printf("\n%s is an identifier",str);  
}  
void main()  
{  
 FILE \*f1,\*f2,\*f3;  
 char c,str[10],st1[10];  
 int num[100],lineno=0,tokenvalue=0,i=0,j=0,k=0;  
 f1=fopen("input","r");  
 f2=fopen("identifier","w");  
 f3=fopen("specialchar","w");  
 while((c=getc(f1))!=EOF)  
 {  
  if(isdigit(c))  
  {  
  tokenvalue=c-'0';  
  c=getc(f1);  
  while(isdigit(c))  
  {  
   tokenvalue\*=10+c-'0';  
   c=getc(f1);  
  }  
  num[i++]=tokenvalue;  
  ungetc(c,f1);  
  }  
  else  
  if(isalpha(c))  
  {  
   putc(c,f2);  
   c=getc(f1);  
   while(isdigit(c)||isalpha(c)||c=='\_'||c=='$')  
   {  
    putc(c,f2);  
    c=getc(f1);  
   }  
   putc(' ',f2);  
   ungetc(c,f1);  
  }  
  else  
  if(c==' '||c=='\t')  
  printf(" ");  
  else  
  if(c=='\n')  
  lineno++;  
  else  
  putc(c,f3);  
 }  
 fclose(f2);  
 fclose(f3);  
 fclose(f1);  
 printf("\n the no's in the program are:");  
 for(j=0;j<i;j++)  
  printf("\t%d",num[j]);  
 printf("\n");  
 f2=fopen("identifier","r");  
 k=0;  
 printf("the keywords and identifier are:");  
 while((c=getc(f2))!=EOF)  
 if(c!=' ')  
 str[k++]=c;  
 else  
 {  
  str[k]='\0';  
  keyword(str);  
  k=0;  
 }  
 fclose(f2);  
 f3=fopen("specialchar","r");  
 printf("\n Special Characters are");  
 while((c=getc(f3))!=EOF)  
 printf("\t%c",c);  
 printf("\n");  
 fclose(f3);  
 printf("Total no of lines are:%d",lineno);  
}



**Ex. No. 2 IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL**

**AIM :**

To write a LEX program to implement a Lexical Analyzer.

**ALGORITHM :**

**Step 1:** Start the program.

**Step 2:** Write regular expression for identifier as **[a-zA-Z][a-zA-Z0-9]\*** and

declare the action as **printf(“identifier”).**

**Step 3:** Write regular expression for preprocessor directive as **#.\***and declare the action

as **printf(“ preprocessor directive”).**

**Step 4:** Write the regular expression for all the keywords and action and declare the

action as **printf(“keywors”).**

**Step 5 :** Write regular expression for all the operators and declare the action as

**printf(“Operators”).**

**Step 6:** Write regular expression for block begins and ends and declare the action

as **printf(“Block begins”)** and **printf(“Block ends”).**

**Step 7:** Write the regular expression for numbers as **[0-9]+** and declare the action as

**printf(“number”).**

**Step 8:** Write the regular expression for string and declare the action as **printf(“string”)**

**Step 9:** Read the input C program **inp.c.**

**Step 10:** Call the function **yylex()** function.

**Step 11:** Stop the program.

**PROGRAM :**

%{

%}

identifier[a-zA-Z][a-zA-Z0-9]\*

%%

#.\* {printf("\n%s is a preprocessor directive",yytext);}

int |

float |

char |

double |

while |

do |

if |

break |

continue |

void |

switch |

return |

else |

goto {printf("\n%s is a keyword",yytext);}

{identifier}\( {printf("\n function %s",yytext);}

\{ {printf("\nblock begins");}

\} {printf("\nblock ends");}

\( {printf("\n");ECHO;}

{identifier}(\[[0-9]\*\])\* {printf("\n%s is an identifier",yytext);}

\".\*\" {printf("\n %s is a string ",yytext);}

[0-9]+ {printf("\n%s is a number",yytext);

}

\<= |

\>= |

\< |

\> |

\== {printf("\n %s is a relational operator",yytext);}

\= |

\+ |

\- |

\/ |

\& |

% {printf("\n %s is a operator",yytext);}

. |

\n;

%%

int main(int argc,char \*\*argv)

{

FILE \*file;

file=fopen("inp.c","r");

if(!file)

{

printf("could not open the file!!!");

exit(0);

}

yyin=file;

yylex();

printf("\n\n");

return(0);

}

int yywrap()

{

return 1;

}

**INPUT FILE :**

#include<stdio.h>

void main()

{

int a,b,c;

printf("enter the value for a,b");

scanf("%d%d",&a,&b)';

c=a+b;

printf("the value of c:%d",&c);

}

**OUTPUT:**

[admin@localhost ~]$ lex lexicalanalyzer.lex

[admin@localhost ~]$ cc lex.yy.c

[admin@localhost ~]$ ./a.out

#include<stdio.h> is a preprocessor directive

void is a keyword

function main(

block begins

int is a keyword

a is an identifier

b is an identifier

c is an identifier

function printf(

"enter the value for a,b" is a string

function scanf(

"%d%d" is a string

& is a operator

a is an identifier

& is a operator

b is an identifier

c is an identifier

= is a operator

a is an identifier

+ is a operator

b is an identifier

function printf(

"the value of c:%d" is a string

& is a operator

c is an identifier

block ends

**RESULT :**

Thus the LEX program to implement lexical analyzer was written and executed successfully.

**Ex. No. 3(a) PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATOR +, - , \* AND /**

**AIM :**

To write a Yacc program to recognize a valid arithmetic expression that uses operator +, - , \* and /.

**ALGORITHM :**

**Step 1:** Start the program

**Step 2:** Declare **num** and **let** be tokens

**Step 3:** Declare the associativity of + and – be left.

**Step 4:** Declare the associativity of \* and / be left.

**Step 5:** Write the grammar for expression as

**expr:  num  
    |  let  
    |  expr '+' expr  
    |  expr '-' expr  
    |  expr '\*' expr  
    |  expr '/' expr  
    |  '(' expr ')'**

Step6: Call **yyparse()** to parse the given expression.

Step7: Override **yylex()** function to check whether the given input is number or letter.

Step 8: Stop the Program.

**PROGRAM :**

%{  
     #include<stdio.h>  
     #include<ctype.h>  
     #include<stdlib.h>  
%}  
%token num let  
%left '+' '-'  
%left '\*' '/'  
%%  
stmt:  stmt '\n'     {printf("\n..valid Expression..\n"); exit(0);}  
    |  expr  
    |  
    |  error '\n'    {printf("\n..Invalid..\n"); exit(0);}  
    ;  
expr:  num  
    |  let  
    |  expr '+' expr  
    |  expr '-' expr  
    |  expr '\*' expr  
    |  expr '/' expr  
    |  '(' expr ')'  
%%  
main()  
{  
  printf("Enter an expression to validate :");  
  yyparse();  
}  
yylex()  
{  
  int ch;  
  while((ch=getchar())==' ');  
  if(isdigit(ch))  
     return num;  
  if(isalpha(ch))  
     return let;  
  return ch;  
}  
yyerror(char \*s)  
{  
  printf("%s",s);  
} 

**OUTPUT :**

[admin@localhost ~]$ yacc -d arithmetic.y

[admin@localhost ~]$ cc y.tab.c

[admin@localhost ~]$ ./a.out

Enter an expression to validate :(a+b)

..valid Expression..

[admin@localhost ~]$ ./a.out

Enter an expression to validate :(a+

syntax error

..Invalid..

[admin@localhost ~]$ ((a+b\*c))

[admin@localhost ~]$ ./a.out

Enter an expression to validate :((a+b\*c))

..valid Expression..

**RESULT :**

Thus the YACC program to recognize a valid arithmetic expression was written and executed successfully.

**Ex. No. 3(b) PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS**

**AIM :**

To write a Yacc program to recognize a valid variable, which starts with a letter,

followed by any number of letters or digits.  
  
**ALGORITHM :**

**Step 1 :** Start the program

**Step 2 :** Declare digand **let** be tokens

**Step3 :** Write the grammar for term as

**TERM:  XTERM '\n'       {printf("\nAccepted\n"); exit(0);}  
    |  error            {yyerror ("Rejected\n"); exit(0);}**

**Step4:** Write the grammar for XTERM as

**XTERM: XTERM let  
      | XTERM dig  
       | let  
Step5:** Call yyparse() to parse the given expression.

**Step6:** Override yylex() function to check whether the given input is digit or letter.

**Step 7:** Stop the Program.

**PROGRAM :**

%{  
   #include<stdio.h>  
   #include<ctype.h>  
   #include<stdlib.h>  
%}  
%token let dig  
%%  
TERM:  XTERM '\n'       {printf("\nAccepted\n"); exit(0);}  
    |  error            {yyerror ("Rejected\n"); exit(0);}  
    ;  
XTERM: XTERM let  
     | XTERM dig  
     | let  
     ;  
%%  
main()  
{  
  printf("Enter a variable:");  
  yyparse();  
}  
yylex()  
{  
 char ch;  
 while((ch=getchar())==' ');  
 if(isalpha(ch))  
    return let;  
 if(isdigit(ch))  
    return dig;  
 return ch;  
}  
yyerror(char \*s)  
{  
  printf("%s",s);  
}

**OUTPUT :**

[admin@localhost ~]$ yacc -d identifier.y

[admin@localhost ~]$ cc y.tab.c

[admin@localhost ~]$ ./a.out

Enter a variable:aregina

Accepted

[admin@localhost ~]$ ./a.out

Enter a variable:1regina

syntax errorRejected

[admin@localhost ~]$ ./a.out

Enter a variable:regina123

Accepted

**RESULT :**

Thus the Yacc program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits was written and executed successfully.

**Ex. No. 3(c) IMPLEMENTATION OF CALCULATOR USING LEX AND YACC**

**AIM :**

To implement a basic calculator using LEX and YACC.

**ALGORITHM :**

**Step 1 :** Start the program

**Step 2 :** Declare DIGITand LETTER be tokens

**Step 3 :** Declare the associativity of + and – be left.

**Step 4 :** Declare the associativity of \* , % and / be left.

**Step 5 :** Declare the associativity of | be left.

**Step 6 :** Declare the associativity of & left.

**Step 7 :** Declare the associativity of unary minus(UMINUS) be left.

**Step 8:** Write the grammar for accepting list of expression and allowing blank spaces between expressions as

**list: /\*empty \*/**

**| list stat '\n'**

**|list error '\n' { yyerrok;};**

**Step 9:** Write the grammar for **stat**

**stat: expr { printf("%d\n",$1); }**

**| LETTER '=' expr { regs[$1] = $3; };**

**Step 10:** Write the grammar for **expr**

**expr: '(' expr ')'   { $$ = $2; }**

**| expr '\*' expr { $$ = $1 \* $3; }**

**| expr '/' expr { $$ = $1 / $3; }**

**| expr '%' expr { $$ = $1 % $3; }**

**| expr '+' expr { $$ = $1 + 3;}**

**| expr '-' expr { $$ = $1 - $3; }**

**| expr '&' expr { $$ = $1 & $3; }**

**| expr '|' expr { $$ = $1 | $3; }**

**|       '-' expr %prec UMINUS { $$ = -$2; }**

**| LETTER { $$ = regs[$1]; }**

**| number**

**;**

**Step 11:** Write the grammar for **number**

**number: DIGIT {**

**$$ = $1;**

**base = ($1==0) ? 8 : 10;**

**}**

**| number DIGIT { $$ = base \* $1 + $ } ;**

**Step12:** Call **yyparse()** to parse the given expression.

**Step 13:** Write the regular expression and actions for identifiers and numbers in the LEX program.

**Step14:** Stop the program.

**PROGRAM :**

calc.yacc:

%{

#include <stdio.h>

int regs[26];

int base;

%}

%start list

%token DIGIT LETTER

%left '|'

%left '&'

%left '+' '-'

%left '\*' '/' '%'

%left UMINUS /\*supplies precedence for unary minus \*/

%% /\* beginning of rules section \*/

list: /\*empty \*/

|

list stat '\n'

|

list error '\n'

{

yyerrok;

}

;

stat: expr

{

printf("%d\n",$1);

}

|

LETTER '=' expr

{

regs[$1] = $3;

}

;

expr: '(' expr ')'

   {

$$ = $2;

}

|

expr '\*' expr

{

$$ = $1 \* $3;

}

|

expr '/' expr

{

$$ = $1 / $3;

}

|

expr '%' expr

{

$$ = $1 % $3;

}

|

expr '+' expr

{

$$ = $1 + $3;

}

          |

expr '-' expr

{

$$ = $1 - $3;

}

|

expr '&' expr

{

$$ = $1 & $3;

}

|

expr '|' expr

{

$$ = $1 | $3;

}

|

      '-' expr %prec UMINUS

  {

$$ = -$2;

}

|

LETTER

{

$$ = regs[$1];

}

|

number

;

number: DIGIT

{

$$ = $1;

base = ($1==0) ? 8 : 10;

} |

number DIGIT

{

$$ = base \* $1 + $2;

}

;

%%

main()

{

return(yyparse());

}

yyerror(s)

char \*s;

{

fprintf(stderr, "%s\n",s);

}

yywrap()

{

return(1);

}

**Calc.lex:**

%{

#include <stdio.h>

#include "y.tab.h"

int c;

extern int yylval;

%}

%%

" " ;

[a-z] {

c = yytext[0];

yylval = c - 'a';

return(LETTER);

}

[0-9] {

c = yytext[0];

yylval = c - '0';

return(DIGIT);

}

[^a-z0-9\b] {

c = yytext[0];

return(c);

}

**OUTPUT :**

[admin@localhost ~]$ yacc -d calc.yacc

[admin@localhost ~]$ lex calc.lex

[admin@localhost ~]$ cc y.tab.c lex.yy.c

[admin@localhost ~]$./a.out

M=4

M+5

9

**RESULT :**

Thus the program to implement a basic calculator using LEX and YACC was written and executed successfully.

**Ex. No.4 GENERATE THREE ADDRESS CODE USING LEX AND YACC**

**AIM :**

To write a LEX and YACC program to convert the given BNF rules to YACC form and write code to generate abstract syntax tree.

**ALGORITHM :**

**Step 1:** Start the program.

**Step 2:** Write the EX program.

**Step2.1:** Write regular expression for identifier as [a-zA-Z][\_a-zA-Z0-9]\*

**Step2.2:** Write regular expression for number as [0-9]+|([0-9]\*\.[0-9]+)

**Step2.3:** If the input string is a keyword, then return keyword. Else copy **yytext** to

**yylval.var**

**Step 2.4:** If the input string is \n then increment the variable **LineNo** by 1.

**Step2.5: If** the input string is a blank or tab space, then perform no action.

**Step 3:** Write the YACC program.

**Step3.1**: Create a structure called **quad** with the elements **op, arg1, arg2** and **result.**

**Step3.2: Create** a structure called stack **with** the elements **items** and **top.**

**Step3.3:** Write grammar for **BLOCK, CODE, STATEMENT, and EXPRESSION.**

**Step3.4:** Define the proper actions to be taken .

**Step3.5:** In the main function call **yyparse()** function .

**Step4:** Stop the program.

**PROGRAM :**

**PROGRAM:**

**LEX PART: ex4.l**

%{

#include<stdio.h>

#include "ex4.tab.h"

%}

%%

[0-9]+ {yylval=atoi(yytext);return NUM;}

[\t] ;

\n {return EOL;}

[-+\*/()] {return yytext[0];}

. {fprintf(stderr,"Error:Invalid Character\n");}

%%

int yywrap(){

return 1;

}

**YACC PART: ex4.y**

%{

#include<stdio.h>

#include<stdlib.h>

int temp\_count=0;

void yyerror(const char\*s){

fprintf(stderr,"Error:%s\n",s);

}

%}

%token NUM EOL

%left '+' '-'

%left '\*' '/'

%%

program:lines

;

lines:lines line

| line

;

line:expr EOL

4931\_GRACE College of Engineering,Thoothukudi

CS3501\_CD

{

printf("Result:t%d\n",$1);

}

;

expr:NUM{

$$=$1;

}

| '(' expr ')'

{

$$=$2;

}

| expr '+' expr

{

printf("t%d=%d+%d\n",++temp\_count,$1,$3);

$$=temp\_count;

}

| expr '-' expr

{

printf("t%d=%d-%d\n",++temp\_count,$1,$3);

$$=temp\_count;

}

| expr '\*' expr

{

printf("t%d=%d\*%d\n",++temp\_count,$1,$3);

$$=temp\_count;

}

| expr '/' expr

{

if($3==0)

{yyerror("Division by zero");

4931\_GRACE College of Engineering,Thoothukudi

CS3501\_CD

$$=0;}

else{

printf("t%d=%d/%d\n",++temp\_count,$1,$3);

$$=temp\_count;

}

}

;

%%

int main()

{

yyparse();

return 0;

}

**OUTPUT:**

2\*10/2+5-1

t1=2\*10

t2=1/2

t3=2+5

t4=3-1

Result:t4

**RESULT:**

Thus the program for Generate three address code for a simple program using LEX and

YACC was executed successfully.

**Ex. No. 5 IMPLEMENTATION OF TYPE CHECKING**

**AIM:**

 To write a C program to implement type checking

**ALGORITHM:**

**Step1:** Track the global scope type information (e.g. classes and their members)

**Step2:** Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

**Step3:** If type found correct, do the operation

**Step4:** Type mismatches, semantic error will be notified

PROGRAM CODE:

//*To implement type checking*

#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i,k,flag=0;

char vari[15],typ[15],b[15],c;

printf("Enter the number of variables:");

scanf(" %d",&n);

for(i=0;i<n;i++)

{

printf("Enter the variable[%d]:",i);

scanf(" %c",&vari[i]);

printf("Enter the variable-type[%d](float-f,int-i):",i);

scanf(" %c",&typ[i]);

if(typ[i]=='f')

flag=1;

}

printf("Enter the Expression(end with $):");

i=0;

getchar();

while((c=getchar())!='$')

{

b[i]=c;

i++;  }

k=i;

for(i=0;i<k;i++)

{

if(b[i]=='/')

{

flag=1;

break;  }  }

for(i=0;i<n;i++)

{

if(b[0]==vari[i])

{

if(flag==1)

{

if(typ[i]=='f')

{  printf("\nthe datatype is correctly defined..!\n");

break;  }

else

{  printf("Identifier %c must be a float type..!\n",vari[i]);

break;  }  }

else

{  printf("\nthe datatype is correctly defined..!\n");

break;  }  }

}

return 0;

}

**RESULT :**

Thus the program was executed and the output was verified successfully.

**Ex.No. : 6 IMPLEMENTATION OF SIMPLE CODE OPTIMIZATION TECHNIQUES**

**AIM :**

To write a C program to implement Code Optimization Techniques.

**ALGORITHM :**

**Step1:** Start the program.

**Step2:** Define a structure with two character fields’ **l** and **r**.

**Step3:** Read the number of statements in **n.**

**Step4**: Store the left value and right value in the structure **op.**

**Step5:** Check for dead code (code that has no further use) and eliminate the same.

**Step 5:** Check for common sub expression and eliminate the same**.**

**Step 6:** Print the optimized code**.**

**Step 7:** Stop the program**.**

**PROGRAM :**

#include<stdio.h>

#include<conio.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;

clrscr();

printf("Enter the Number of Values:");

scanf("%d",&n);

for(i=0;i<n;i++)

{

printf("left: ");

op[i].l=getche();

printf("\tright: ");

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",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';

strcpy(pr[i].r,'\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);

}

}

getch();

}

**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 El

imination

b =c+d

e =c+d

f =b+e

r =:f

Eliminate Common E

xpression

b =c+d

b =c+d

f =b+b

r =:f

Optimized C

ode

b=c+d

f=b+b

r=:f

**RESULT :**

Thus the C program to implement code optimization was written and successfully executed.

**Ex. No. 7 IMPLEMENTATION OF BACK END OF A COMPILER**

**AIM :**

To write a C program to implement back end of a compiler that uses simple instructions like move, add, sub.

**ALGORITHM :**

**Step1:** Start the program.

**Step2:** Open the file that has the intermediate code as input.

**Step3:** Repeat Step 4 to until end of file is reached.

**Step4**: Read the intermediate code from the file

**Step5:** If the statement uses assignment operator, then generate the code with MOV instruction.

If + is used in a statement then then generate the code with ADD instruction

If - is used in a statement then then generate the code with SUB instruction.

**Step 5:** Print the generated assembly code**.**

**Step 6:** Stop the program**.**

**PROGRAM :**

#include<stdio.h>

#include<conio.h>

#include<ctype.h>

#include<stdlib.h>

void main()

{

inti=2,j=0,k=2,k1=0;

char ip[10],kk[10];

FILE \*fp;clrscr();

printf("\nEnter the filename of the intermediate code");

scanf("%s",&kk);

fp=fopen(kk,"r");

if(fp==NULL)

{

printf("\nError in Opening the file");

getch();

}

clrscr();

while(!feof(fp))

{fscanf(fp,"%s\n",ip);

printf("\t\t%s\n",ip);

}

rewind(fp);

printf("\n------------------------------\n");

printf("\tStatement \t\t target code\n");

printf("\n------------------------------\n");

while(!feof(fp))

{

fscanf(fp,"%s",ip);

printf("\t%s",ip);

printf("\t\tMOV %c,R%d\n\t",ip[i+k],j);

if(ip[i+1]=='+')

printf("\t\tADD");

elseprintf("\t\tSUB");

if(islower(ip[i]))

printf("%c,R%d\n\n",ip[i+k1],j);

else

printf("%c,%c\n",ip[i],ip[i+2]);

j++;

k1=2;

k=0;

}

printf("\n-------------------------------\n");

getch();

fclose(fp);

}

**INPUT : k.txt**

X=a-b

Y=a-c

z=a+b

C=a-b

C=a-b

**OUTPUT :**

Enter the filename of the intermediate code k.txt

X=a-b

Y=a-c

z=a+b

C=a-b

C=a-b

------------------------------Statement target code-----------------------------

-X=a-b MOV b,R0SUBa,R0

Y=a-c MOV a,R1SUBc,R1

z=a+b MOV a,R2ADDb,R2

C=a-b MOV a,R3SUBb,R3

**RESULT :**

Thus the program to implement the backend of a compiler is written and executed.