19CSE401 – Compiler Design

Index

Name: Aman Kumar Rauniyar

Reg. No: CH.EN.U4CSE22174

Lab Exercise: 01

Basic Lex Programs

1.Title: Write a program to check if a given number is prime or not.

Code:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%%
[0-9]+    {
            int num = atoi(yytext);

            if(num <= 1) {
               printf("%d is not prime.\n", num);
            } else {
               int i, flag = 1;
               int limit = (int)sqrt(num);

               for(i = 2; i <= limit; i++) {
                  if(num % i == 0) {
                     flag = 0;
                     break;
                  }
               }
               if(flag)
                  printf("%d is prime.\n", num);
               else
                  printf("%d is not prime.\n", num);
            }
         }
\n    ;   // ignore new lines
.      ;   // ignore other characters
%%
```

```
int main() {
    printf("Enter a number: ");
    yylex();
    return 0;
}

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

<u>Output:</u>

```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ flex prime.l
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ gcc lex.yy.c -ll -lm -o prime_check
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ ./prime_check
Enter a number: 1
1 is not prime.

asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ ./prime_check
Enter a number: 2
2 is prime.
```

2.Title: Write a program to reverse a string without using built-in functions.

<u>Code:</u>

```
%{
#include <stdio.h>

void reverse(char *str, int length) {
    int i;
    for(i = 0; i < length / 2; i++) {
        char temp = str[i];
        str[i] = str[length - 1 - i];
        str[length - 1 - i] = temp;
    }
}

%}

%%
.*\n    {
            // yytext contains the whole line including newline
            int length = 0;
            // Calculate length excluding newline
            while(yytext[length] != '\n' && yytext[length] != '\0') {
```

```
            length++;
        }

        // Reverse the string in yytext (modifying in place)
        reverse(yytext, length);

        // Add newline back manually
        yytext[length] = '\n';
        yytext[length+1] = '\0';

        printf("Reversed string: %s", yytext);
        return 0; // Stop after processing one line
    }
%%
int main() {
    printf("Enter a string: ");
    yylex();
    return 0;
}

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

Output:

```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ flex reverse.l
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ gcc lex.yy.c -ll -o reverse_string
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ ./reverse_string
Enter a string: Aashutosh Kumar Pandit
Reversed string: tidnaP ramuK hsotuhsaA
```

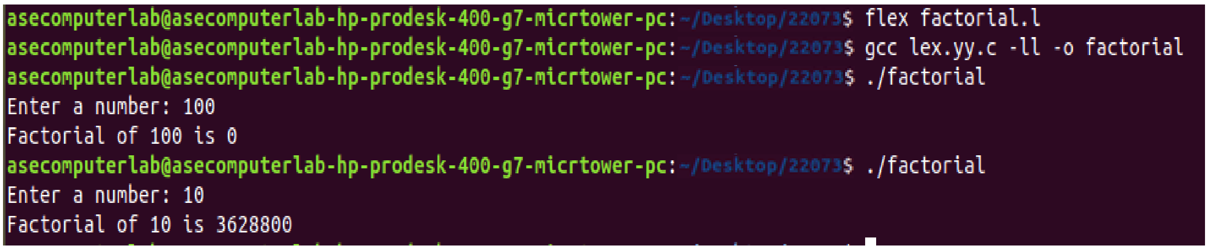
3.Title: Write a program to find the factorial of a number using recursion.

Code:

```
%{
#include <stdio.h>

// Recursive factorial function
long long factorial(int n) {
    if (n <= 1)
```

```
        return 1;
    else
        return n * factorial(n - 1);
}
%}

%%
[0-9]+  {
            int num = atoi(yytext);
            printf("Factorial of %d is %lld\n", num, factorial(num));
            return 0;  // Stop after processing one number
        }
\n      ;  // ignore newline
.       ;  // ignore any other characters
%%

int main() {
    printf("Enter a number: ");
    yylex();
    return 0;
}

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

Output:



4.Title: Write a program to find the largest and smallest element in an array.

Code:

```
%{
#include <stdio.h>
#include <limits.h>
```

```
int largest = INT_MIN;
int smallest = INT_MAX;
%}

%%
[0-9]+ {
            int num = atoi(yytext);
            if (num > largest)
                largest = num;
            if (num < smallest)
                smallest = num;
        }
[\n\t ]+  ;  // Ignore whitespace including newlines, tabs, spaces

.       ;  // Ignore any other characters
%%

int main() {
    printf("Enter numbers separated by space (Ctrl+D or Ctrl+Z to end input):\n");
    yylex();
    printf("Largest element: %d\n", largest);
    printf("Smallest element: %d\n", smallest);
    return 0;
}

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

<u>Output:</u>

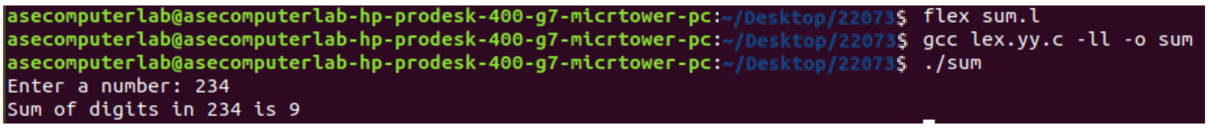
```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ flex ls.l
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ gcc lex.yy.c -ll -o ls
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/22073$ ./ls
Enter numbers separated by space (Ctrl+D or Ctrl+Z to end input):
12 7 8 100 92 78 26
Largest element: 100
Smallest element: 7
```

5.Title: Write a program to find the sum of digits of a given number.

Code:

```
%{
#include <stdio.h>
%}

%%
[0-9]+ {
            int sum = 0;
            char *p = yytext;
            while (*p) {
                sum += (*p - '0');  // convert char digit to int and add
                p++;
            }
            printf("Sum of digits in %s is %d\n", yytext, sum);
            return 0;  // stop after processing one number
        }
\n     ;  // ignore newlines
.      ;  // ignore other characters
%%

int main() {
    printf("Enter a number: ");
    yylex();
    return 0;
}

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

Output:



Name: Aman Kumar Rauniyar

Reg.No.:CH.EN.U4CSE22174

Experiment No: 02

Aim: To implement eliminate left recursion and left factoring from the given grammar using C program.

i. Left factoring

Code:

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

int main() {
    char gram[100], part1[100], part2[100], modifiedGram[100], newGram[100];
    int i, j = 0, k = 0, pos = 0;

    printf("Enter Production : A->");
    gets(gram);  // Note: unsafe, consider fgets for real code

    // Split input at '|'
    for (i = 0; gram[i] != '|' && gram[i] != '\0'; i++, j++)
        part1[j] = gram[i];
    part1[j] = '\0';

    for (j = i + 1, i = 0; gram[j] != '\0'; j++, i++)
        part2[i] = gram[j];
    part2[i] = '\0';

    // Find common prefix
    for (i = 0; i < strlen(part1) && i < strlen(part2); i++) {
        if (part1[i] == part2[i]) {
            modifiedGram[k++] = part1[i];
            pos = i + 1;
        } else
            break;  // stop at first mismatch
    }

    // Build new production after factoring
    for (i = pos, j = 0; part1[i] != '\0'; i++, j++)
        newGram[j] = part1[i];
    newGram[j++] = '|';

    for (i = pos; part2[i] != '\0'; i++, j++)
        newGram[j] = part2[i];

    modifiedGram[k++] = 'X';  // new variable for factoring
    modifiedGram[k] = '\0';
    newGram[j] = '\0';

    printf("\n A->%s", modifiedGram);
    printf("\n X->%s\n", newGram);

    return 0;
}
```

Output:

```
ubuntu:~$ gcc ex2.c
ex2.c: In function ‘main’:
ex2.c:9:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets
’? [-Wimplicit-function-declaration]
     gets(gram);  // Note: unsafe, consider fgets for real code
     ^~~~
     fgets
/tmp/ccnIWJvK.o: In function `main':
ex2.c:(.text+0x5a): warning: the `gets' function is dangerous and should not be
used.
ubuntu:~$ ./a.out
Enter Production : A->aE+X

 A->X
 X->aE+X|
ubuntu:~$
```

ii. Left Recursion

Code:

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

#define SIZE 100

int main() {
    char non_terminal;
    char beta, alpha;
    int num;
    char production[10][SIZE];
    int index;

    printf("Enter Number of Productions: ");
    scanf("%d", &num);

    printf("Enter the grammar productions (e.g. E->E-A):\n");
    for (int i = 0; i < num; i++) {
        scanf("%s", production[i]);
    }

    for (int i = 0; i < num; i++) {
        printf("\nGRAMMAR: %s", production[i]);

        non_terminal = production[i][0];
        index = 3; // position after '->'

        if (production[i][index] == non_terminal) {
            alpha = production[i][index + 1];
            printf(" is left recursive.\n");

            // Move index forward to the end of alpha part (before '|')
            while (production[i][index] != '\0' && production[i][index] != '|') {
                index++;
            }
```

```c
                }

                if (production[i][index] == '|') {
                    beta = production[i][index + 1];
                    printf("Grammar without left recursion:\n");
                    printf("%c->%c%c'\n", non_terminal, beta, non_terminal);
                    printf("%c'->%c%c'|ε\n", non_terminal, alpha, non_terminal);
                } else {
                    printf(" can't be reduced\n");
                }
            } else {
                printf(" is not left recursive.\n");
            }
        }

        return 0;
    }
```

Output:

```
ubuntu:~$ gedit lab2.1.c
ubuntu:~$ gcc lab2.1.c
ubuntu:~$ ./a.out
Enter Number of Productions: 2
Enter the grammar productions (e.g. E->E-A):
E->A/B
eX+B

GRAMMAR: E->A/B is not left recursive.

GRAMMAR: eX+B is not left recursive.
ubuntu:~$
```

Results: The program to implement left factoring and left recursion has been successfully executed.

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Experiment No: 03

Aim: To implement LL(1) parsing using C program.

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

char s[20], stack[20];

// Parsing table for predictive parsing (non-terminal x terminal)
char *m[5][6] = {
    /* i      +       *       (       )       $   */
    {"tb",  "",     "",     "tb",   "",     ""},   // e
    {"",    "+tb",  "",     "",     "n",    "n"},  // b
    {"fc",  "",     "",     "fc",   "",     ""},   // t
    {"",    "n",    "*fc",  "",     "n",    "n"},  // c
    {"i",   "",     "",     "(e)",  "",     ""}    // f
};

int size[5][6] = {
    {2, 0, 0, 2, 0, 0},  // e
    {0, 3, 0, 0, 1, 1},  // b
    {2, 0, 0, 2, 0, 0},  // t
    {0, 1, 3, 0, 1, 1},  // c
    {1, 0, 0, 3, 0, 0}   // f
};

int main()
{
    int i, j, k;
    int str1, str2;
    int n;

    printf("\nEnter the input string: ");
    scanf("%s", s);

    strcat(s, "$");
    n = strlen(s);

    stack[0] = '$';
    stack[1] = 'e';
    i = 1; // top of stack index
    j = 0; // input pointer index

    printf("\nStack\tInput\n");
    printf("______\n\n");

    // Continue until BOTH stack top and input symbol are '$'
```

Code:

```c
    // Continue until BOTH stack top and input symbol are '$'
    while (!(stack[i] == '$' && s[j] == '$')) {
        if (stack[i] == s[j]) {
            // Match terminal
            i--;
            j++;
        } else {
            // Get row for non-terminal on top of stack
            switch (stack[i]) {
                case 'e': str1 = 0; break;
                case 'b': str1 = 1; break;
                case 't': str1 = 2; break;
                case 'c': str1 = 3; break;
                case 'f': str1 = 4; break;
                default:
                    printf("\nERROR: Invalid non-terminal %c\n", stack[i]);
                    exit(0);
            }

            // Get column for current input symbol
            switch (s[j]) {
                case 'i': str2 = 0; break;
                case '+': str2 = 1; break;
                case '*': str2 = 2; break;
                case '(': str2 = 3; break;
                case ')': str2 = 4; break;
                case '$': str2 = 5; break;
                default:
                    printf("\nERROR: Invalid input symbol %c\n", s[j]);
                    exit(0);
            }

            if (m[str1][str2][0] == '\0') {
                printf("\nERROR: No rule for [%c][%c]\n", stack[i], s[j]);
                exit(0);
            } else if (m[str1][str2][0] == 'n') {
                // 'n' means epsilon production (pop)
                i--;
            } else if (m[str1][str2][0] == 'i') {
                // 'i' means push 'i' on stack
                stack[i] = 'i';
            } else {
                // Push RHS of production in reverse order
                for (k = size[str1][str2] - 1; k >= 0; k--) {
```


```c
            } else {
                // Push RHS of production in reverse order
                for (k = size[str1][str2] - 1; k >= 0; k--) {
```


```c
                    stack[i] = m[str1][str2][k];
                    i++;
                }
                i--; // Adjust for extra increment
            }
        }

        // Print stack
        for (k = 0; k <= i; k++)
            printf("%c", stack[k]);
        printf("\t");

        // Print input from current pointer
        for (k = j; k < n; k++)
            printf("%c", s[k]);
        printf("\n");
    }

    if (stack[i] == '$' && s[j] == '$')
        printf("\nSUCCESS\n");
    else
        printf("\nERROR: Parsing incomplete\n");

    return 0;
}
```

Output:

```
ubuntu:~$ gcc third.c
ubuntu:~$ ./a.out
```

```
Enter the input string: i*i+i

Stack   Input
______

$bt     i*i+i$
$bcf    i*i+i$
$bci    i*i+i$
$bc     *i+i$
$bcf*   *i+i$
$bcf    i+i$
$bci    i+i$
$bc     +i$
$b      +i$
$bt+    +i$
$bt     i$
$bcf    i$
$bci    i$
$bc     $
$b      $
$       $

SUCCESS
ubuntu:~$
```

Results:

The program to implement left factoring and left recursion has been successfully executed.

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Experiment No: 04

Aim: To write a program in YACC for parser generation.

```
%{
#include <stdio.h>
#include <ctype.h>
#define YYSTYPE double

int yylex();
int yyerror(const char *s);
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:
    lines expr '\n' {
        printf("%g\n", $2);
    }
  | lines '\n'
  | /* empty */
;
expr:
    expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
  | expr '*' expr { $$ = $1 * $3; }
  | expr '/' expr { $$ = $1 / $3; }
  | '-' expr %prec UMINUS { $$ = -$2; }
  | '(' expr ')' { $$ = $2; }
  | NUMBER
;
%%
int yylex() {
    int c;

    // Skip whitespace
    while ((c = getchar()) == ' ' || c == '\t');

    if (c == '.' || isdigit(c)) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }    return c;
}
int yyerror(const char *s) {
```

```
}
int yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 1;
}
int main() {
    return yyparse();
}
int yywrap() {
    return 1;
}
```

Code:

```
ubuntu:~$ yacc 4.y
ubuntu:~$ gcc -o 4 y.tab.c
ubuntu:~$ ./4
20+51
71
11+22
33
3463846+373623
3.83747e+06
323-121
202
3212+1616
4828
22074+22078
44152
```

Output:

Results:

The program in YACC for parser generation has been executed successfully

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Lab Exp.: 05

Aim: To implement symbol table.

<u>Code:</u>

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

int main() {
  int x = 0, i = 0, j = 0;
  void *T4Tutorials_address[50];   // Symbol addresses
  char T4Tutorials_Array2[50];     // Input expression
  char T4Tutorials_Array3[50];     // Symbols stored
  char c;

  printf("Input the expression ending with $ sign: ");
  while ((c = getchar()) != '$') {
    T4Tutorials_Array2[i++] = c;
  }
  int n = i - 1;

  // Display the entered expression
  printf("\nGiven Expression: ");
  for (i = 0; i <= n; i++) {
    printf("%c", T4Tutorials_Array2[i]);
  }

  // Display Symbol Table
  printf("\n\nSymbol Table display\n");
  printf("Symbol \t Address \t Type\n");

  for (j = 0; j <= n; j++) {
    c = T4Tutorials_Array2[j];
    if (isalpha(c)) {
      // Allocate memory for identifier (1 byte per char)
      void *mypointer = malloc(sizeof(char));
      T4Tutorials_address[x] = mypointer;
```

```
            T4Tutorials_Array3[x] = c;
            printf("%c \t %p \t identifier\n", c, mypointer);
            x++;
        } else if (c == '+' || c == '-' || c == '*' || c == '=') {
            // Allocate memory for operator (1 byte)
            void *mypointer = malloc(sizeof(char));
            T4Tutorials_address[x] = mypointer;
            T4Tutorials_Array3[x] = c;
            printf("%c \t %p \t operator\n", c, mypointer);
            x++;
        }
    }

    // Free allocated memory
    for (i = 0; i < x; i++) {
        free(T4Tutorials_address[i]);
    }

    return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ bison -d calc.y
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ gcc -o calc calc.tab.c -lm
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ ./calc
1 + 2 * 3
(4 + 5) / 3
-6 + 7
7
3
1
```

Result: Thus, the program to implement symbol table has been executed successfully.

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Lab Exp.: 06

Aim: To implementation of intermediate code generation.

Code:

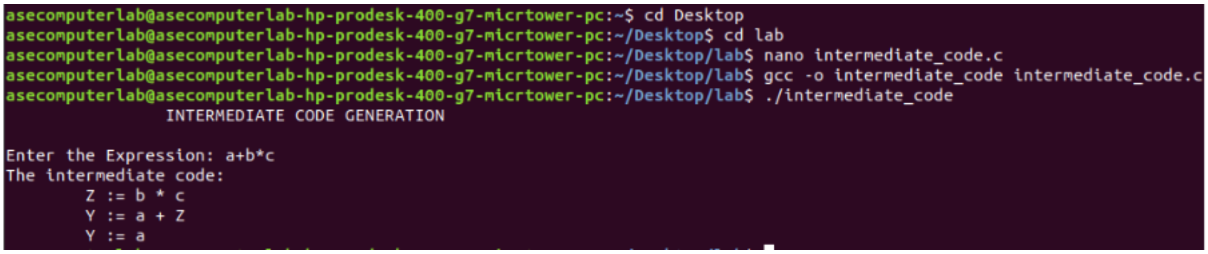
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Global variables
int i = 1, j = 0, no = 0, tmpch = 90; // tmpch = 90 corresponds to 'Z'
char str[100], left[15], right[15];

// Structure for expression components
struct exp {
   int pos;
   char op;
} k[15]; // Array of structs to hold operators and their positions

// Function prototypes
void findopr();
void explore();
void fleft(int);
void fright(int);

// Main function
int main() {
   printf("\t\tINTERMEDIATE CODE GENERATION\n\n");
   printf("Enter the Expression: ");
   scanf("%s", str); // Read the expression into str
   printf("The intermediate code:\n");

   findopr(); // Identify and store operators
   explore(); // Generate intermediate code

   return 0;
}
```

```
// Function to explore the operators and generate code
void explore() {
   int i = 0;
   // Loop through the stored operators until a null character is found in the op field
   while (k[i].op != '\0') {
      // Clear left and right strings for the current operation
      fleft(k[i].pos);
      fright(k[i].pos);

      // Assign a temporary variable name (starting from 'Z' and decrementing)
      str[k[i].pos] = tmpch--;

      // Print the three-address code statement
      printf("\tT%c := %s %c %s\n", str[k[i].pos], left, k[i].op, right);
      i++;
   }

   // Process the final result after all operations are reduced
   fright(-1); // Get the final expression (which should be a single character/variable)
   if (no == 0) {
      // If no operators were processed (i.e., it was a single operand)
      fleft(strlen(str));
      printf("\tT%s := %s\n", right, left);
      exit(0); // Exit the program
   }

   // Print the final assignment
   printf("\tT%c := %s\n", right, str[k[-i].pos]); // Note: k[-i] seems like a likely typo in the image, maybe it
should be k[i-1] or a simple variable like T0.
                                        // Based on the surrounding logic, it seems to be accessing the final temporary
variable name.
                                        // Assuming the original intent was to display the last generated temporary
variable.
}

// Function to find the left operand for the operator at position x in str
void fleft(int x) {
   int w = 0, flag = 0;
   x--; // Start searching one character before the operator

   // Loop backwards from x until an operator, '$' (which indicates a reduced expression), or -1 (start of string)
is found
```

```
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '=' &&
           str[x] != '0' && str[x] != '-' && str[x] != '!' && str[x] != '/' &&
           str[x] != ':') {

        if (str[x] != '$' && flag == 0) {
            left[w++] = str[x]; // Collect the character
            left[w] = '\0';
            str[x] = '$'; // Mark the character as processed (replaced by '$')
            flag = 1;
        }
        x--;
    }

    // Reverse the left string because it is collected backwards
    int start = 0, end = w - 1;
    while (start < end) {
        char temp = left[start];
        left[start] = left[end];
        left[end] = temp;
        start++;
        end--;
    }
}

// Function to find the right operand for the operator at position x in str
void fright(int x) {
    int w = 0, flag = 0;

    // If x is not -1 (meaning it's not the final step)
    if (x != -1) {
        x++; // Start searching one character after the operator
    } else {
        x = 0; // Start from the beginning of the string for the final reduction
    }

    // Loop until an operator or null character is found
    while (x != -1 && str[x] != '\0' && str[x] != '+' && str[x] != '*' &&
           str[x] != '=' && str[x] != ':' && str[x] != '!' && str[x] != '/' &&
           str[x] != '-') {

        if (str[x] != '$' && flag == 0) {
            right[w++] = str[x]; // Collect the character
```

```
        right[w] = '\0';
        str[x] = '$'; // Mark the character as processed (replaced by '$')
        flag = 1;
      }
      x++;
    }
}
```

Output:



```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~$ cd Desktop
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop$ cd lab
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ nano intermediate_code.c
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ gcc -o intermediate_code intermediate_code.c
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ ./intermediate_code
                INTERMEDIATE CODE GENERATION

Enter the Expression: a+b*c
The intermediate code:
        Z := b * c
        Y := a + Z
        Y := a
```

Result: Thus, the program to implement intermediate code generation has been executed successfully.

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Lab Exp.: 07

Aim: To implementation of Code Optimization Techniques.

Code:

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

// Structure to represent an intermediate code statement
struct op {
   char l;    // Left part (the variable being assigned to)
   char r[20]; // Right part (the expression)
} op[10], pr[10]; // op[] stores the original code, pr[] stores the optimized code

int 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); // Read the number of statements (n)

   // Input the intermediate code statements
   for (i = 0; i < n; i++) {
      printf("left: ");
      // Assuming single character variable name on the left side
      scanf(" %c", &op[i].l);
      printf("right: ");
      // Reading the expression on the right side
      scanf("%s", op[i].r);
   }

   printf("\nIntermediate Code\n");
   for (i = 0; i < n; i++) {
      printf("%c = %s\n", op[i].l, op[i].r);
   }
```

```
// Dead code elimination part
// Copy only statements where the assigned variable (op[i].l) is used in a later statement's right side.
for (i = 0; i < n - 1; i++) {
   temp = op[i].l; // Variable assigned in current statement

   // Check if the variable 'temp' is used in any subsequent statement's right side
   for (j = i + 1; j < n; j++) {
      p = strchr(op[j].r, temp); // Search for 'temp' in op[j].r
      if (p) {
         // If found, this statement is NOT dead. Copy it to the 'pr' array.
         pr[z].l = op[i].l;
         strcpy(pr[z].r, op[i].r);
         z++;
         break; // Once found, no need to add duplicates for this statement (op[i])
      }
   }
}

// Add last statement as it is (it's assumed the result of the last statement is used/printed outside)
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 = %s\n", pr[k].l, pr[k].r);
}
// Common subexpression elimination
for (m = 0; m < z; m++) {
   tem = pr[m].r; // Right side of the current statement

   // Compare with all subsequent statements
   for (j = m + 1; j < z; j++) {
      p = strstr(tem, pr[j].r); // Check if pr[j].r is a common subexpression in pr[m].r
      if (p) {
         t = pr[j].l; // Variable assigned by the later common subexpression
         pr[j].l = pr[m].l; // Replace the variable of the later statement with the earlier one

         // The following inner loop seems intended to update the right sides of other statements
         // that might use the later common subexpression (pr[j].l), replacing it with the earlier one (pr[m].l)
         for (i = 0; i < z; i++) {
            l = strchr(pr[i].r, t); // Search for the eliminated variable 't' in pr[i].r
```

```
                if (l) {
                    a = l - pr[i].r; // Position of the character 't'
                    pr[i].r[a] = pr[m].l; // Replace it with the earlier variable 'pr[m].l'
                }
            }
        }
    }
}

printf("\nAfter Eliminating Common Expressions\n");
for (i = 0; i < z; i++) {
    printf("%c = %s\n", pr[i].l, pr[i].r);
}
// Remove duplicates by marking them '\0' (This step cleans up the result of CSE)
for (i = 0; i < z; i++) {
    // Compare statement i with all following statements j
    for (j = i + 1; j < z; j++) {
        // Compare the right sides
        q = strcmp(pr[i].r, pr[j].r);

        // If right sides are the same AND the left sides are the same (e.g., a=b+c and a=b+c)
        if ((pr[i].l == pr[j].l) && (q == 0)) {
            pr[i].l = '\0'; // Mark the earlier duplicate statement's left variable as '\0' for removal
        }
    }
}
printf("\nOptimized Code\n");
for (i = 0; i < z; i++) {
    // Print only statements that haven't been marked for removal
    if (pr[i].l != '\0') {
        printf("%c = %s\n", pr[i].l, pr[i].r);
    }
}
return 0;
}
```

Output:

```
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ nano dead_code_elimination.c
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ gcc -o dead_code dead_code_elimination.c
asecomputerlab@asecomputerlab-hp-prodesk-400-g7-micrtower-pc:~/Desktop/lab$ ./dead_code
Enter the Number of Values: 4
left: a
right: b+c
left: d
right: a+e
left: f
right: d+g
left: h
right: a+i

Intermediate Code
a = b+c
d = a+e
f = d+g
h = a+i

After Dead Code Elimination
a = b+c
d = a+e
h = a+i

After Eliminating Common Expressions
a = b+c
d = a+e
h = a+i

Optimized Code
a = b+c
d = a+e
h = a+i
```

Result: Thus, the program to implement Code Optimization Techniques has been executed successfully.

Name: Aman Kumar Rauniyar

Reg. No.: CH.EN.U4CSE22174

Lab Exp.: 08

Aim: To write a program that implements the target code generation.

Code:

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

// Global variables
int label[20]; // Array to store instruction numbers that are jump targets
int no = 0;    // Counter for the number of labels stored

// Function to check if a given instruction number 'k' is a jump target
int check_label(int k) {
   int i;
   for (i = 0; i < no; i++) {
      if (k == label[i])
         return 1; // It is a jump target
   }
   return 0; // It is not a jump target
}

int main() {
   FILE *fp1, *fp2;
   char fname[10], op[10], ch;
   char operand1[8], operand2[8], result[8];
   int i = 0, j = 0;

   printf("\n Enter filename of the intermediate code: ");
   scanf("%s", fname);

   // Open the intermediate code file for reading and the target file for writing
   fp1 = fopen(fname, "r");
   fp2 = fopen("target.txt", "w");

   if (fp1 == NULL || fp2 == NULL) {
      printf("\n Error opening the file");
```

```
    exit(0);
}

// Process the intermediate code file line by line
while (!feof(fp1)) {
    fprintf(fp2, "\n"); // New line for formatting in the target file
    fscanf(fp1, "%s", op); // Read the operation/opcode

    // Increment the instruction counter
    i++;

    // Check if the current instruction is a target of a previous jump
    if (check_label(i)) {
        fprintf(fp2, "\nlabel#%d:", i); // Print the label
    }

    // --- Specific Operations (using strcmp for multi-character opcodes) ---

    // PRINT operation
    if (strcmp(op, "print") == 0) {
        fscanf(fp1, "%s", result);
        fprintf(fp2, "\n\t OUT %s", result);
    }

    // GOTO operation (Unconditional Jump)
    else if (strcmp(op, "goto") == 0) {
        fscanf(fp1, "%s %s", operand1, operand2); // Reads condition and target instruction number
        fprintf(fp2, "\n\t JMP %s,label#%s", operand1, operand2);
        label[no++] = atoi(operand2); // Store the target instruction number as a label
    }

    // Array assignment: []= (e.g., A[i] = B)
    else if (strcmp(op, "[]=") == 0) {
        fscanf(fp1, "%s %s %s", operand1, operand2, result);
        // Assuming intermediate code is: []= A i B (meaning A[i] = B)
        fprintf(fp2, "\n\t STORE %s[%s],%s", operand1, operand2, result);
    }

    // Unary Minus operation: uminus (e.g., T1 = uminus A)
    else if (strcmp(op, "uminus") == 0) {
        fscanf(fp1, "%s %s", operand1, result); // Reads operand and result
        fprintf(fp2, "\n\t LOAD -%s,R1", operand1); // Load the negative value into R1
```

```
   fprintf(fp2, "\n\t STORE R1,%s", result);    // Store R1 into the result variable
}

// --- Arithmetic and Relational Operations (using switch for single-character opcodes) ---
else {
   switch (op[0]) {
      case '*': // Multiplication: * A B T1 (T1 = A * B)
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         // NOTE: The original code's LOAD line is missing an operand. Correcting to a likely intent.
         // Original: fprintf(fp2,"\n \tLOAD",operand1);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t MUL R1,R0"); // R0 = R0 * R1
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;

      case '+': // Addition: + A B T1 (T1 = A + B)
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t ADD R1,R0"); // R0 = R0 + R1
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;

      case '-': // Subtraction: - A B T1 (T1 = A - B)
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n\t LOAD %s,R0", operand1); // Load A into R0
         fprintf(fp2, "\n \tLOAD %s,R1", operand2); // Load B into R1
         fprintf(fp2, "\n \t SUB R1,R0"); // R0 = R0 - R1 (A - B)
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;

      case '/': // Division: / A B T1 (T1 = A / B)
         // NOTE: The original code has a typo: "%s %s s". Correcting to "%s %s %s".
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t DIV R1,R0"); // R0 = R0 / R1
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;

      case '%': // Modulo (Using DIV instruction, which is often used for MOD/REM)
```

```
            fscanf(fp1, "%s %s %s", operand1, operand2, result);
            fprintf(fp2, "\n \t LOAD %s,R0", operand1);
            fprintf(fp2, "\n \t LOAD %s,R1", operand2);
            fprintf(fp2, "\n \t DIV R1,R0"); // In many architectures, DIV sets a remainder register.
                                    // This code simply uses DIV and stores R0, which is likely incorrect for MOD.
                                    // Sticking to the code's original instruction pattern.
            fprintf(fp2, "\n \t STORE R0,%s", result);
            break;

          case '=': // Assignment: = A T1 (T1 = A)
            fscanf(fp1, "%s %s", operand1, result);
            // NOTE: The instruction STORE is commonly used for this, but the original code is STORE %s %s.
            // Correcting to a more standard pattern: LOAD into a register, then STORE.
            // Sticking to the code's original instruction pattern, assuming it means STORE operand1 to result.
            fprintf(fp2, "\n\t STORE %s, %s", operand1, result);
            break;

          case '>': // Greater Than Conditional Jump: > A B target (If A > B, goto target)
            j++;
            fscanf(fp1, "%s %s %s", operand1, operand2, result); // Reads A, B, and target instruction number
            fprintf(fp2, "\n \t LOAD %s,R0", operand1);       // Load the first operand A into R0
            fprintf(fp2, "\n\t JGT %s,label#%s", operand2, result); // Jump if Greater Than
            label[no++] = atoi(result);
            break;

          case '<': // Less Than Conditional Jump: < A B target (If A < B, goto target)
            fscanf(fp1, "%s %s %s", operand1, operand2, result);
            fprintf(fp2, "\n \t LOAD %s,R0", operand1);
            // NOTE: The original code has a typo: label#%d. Correcting to label#%s to match 'result' being a
string.
            fprintf(fp2, "\n\t JLT %s,label#%s", operand2, result); // Jump if Less Than
            label[no++] = atoi(result);
            break;
        }
      }
    }

    // Close and reopen the target file to read and display the generated code
    fclose(fp2);
    fclose(fp1);

    fp2 = fopen("target.txt", "r");
```

```
    if (fp2 == NULL) {
        printf("Error opening the file\n");
        exit(0);
    }

    // Print the generated target code to the console
    printf("\n\nGenerated Target Code:\n");
    do {
        ch = fgetc(fp2);
        printf("%c", ch);
    } while (ch != EOF);

    fclose(fp2);
    // NOTE: The original code tries to close fp1 again here, which is redundant.

    return 0;
}
```

Output:

```
Enter filename of the intermediate codeinput.txt


        LOAD t2,R0
        LOAD t2,R1
        DIV R1,R0
        STORE R0,�U

        LOAD -t2,R1
        STORE R1,t2

        OUT t2

        LOAD t3,R0
        LOAD t4,R1
        ADD R1,R0
        STORE R0,print
```

Result: Thus, the program to implement the target code generation has been executed successfully.