LAB RECORD
Compiler Design
19CSE401

BONAFIDE CERTIFICATE

University Reg. No : CH.EN.U4CSE22160

This is to certify that this is a bonafide record work done by Mr. / Miss. Velchuri Sarvan studying B.Tech CSE 4th year.

Internal Examiner1 **Internal Examiner2**

Index

Experiment No (LAB)	**Title**	**Page No**
01	Implementation of Lexical Analyser using Lex Tools	4-8
02	Program to eliminate left recursion and factoring from the given grammar	9-11
03	Implementation of LL(1) Parsing	12-14
04	Parser Generation Using YACC	15-17
05	Implementation of Symbol Table	18-19
06	Implementation of Intermediate Code Generation	20-21
07	Implementation of Code Optimization Techniques	22-24
08	Implementation of target code generation	25-26

Lab Exercise -01

1. Aim: To write a program that identifies and counts words in an input sentence.

Program:

```
  GNU nano 2.9.3                              count_words.l

%{
#include <stdio.h>
int words = 0;
%}

%%
[a-zA-Z0-9]+     { words++; }
.|\n             ;  // Ignore everything else
%%

int yywrap() { return 1; }

int main() {
    printf("Enter text (Ctrl+D to end):\n");
    yylex();
    printf("Total words: %d\n", words);
    return 0;
}
```

Output:

```
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano count_words.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ lex count_words.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ gcc lex.yy.c -o count_words -lfl
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ ./count_words
Enter text (Ctrl+D to end):
anindita 22180
Total words: 2
```

2. Aim: To write a program that checks whether a given number is odd or even.

Program:

```
GNU nano 2.9.3                                    odd_even.l

%{
#include <stdio.h>
int num;
%}

%%
[0-9]+ {
    num = atoi(yytext);
    if (num % 2 == 0)
        printf("%d is Even\n", num);
    else
        printf("%d is Odd\n", num);
}
.|\n    ;
%%

int yywrap() { return 1; }

int main() {
    printf("Enter numbers (Ctrl+D to end):\n");
    yylex();
    return 0;
}
```

Output:

```
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano odd_even.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano odd_even.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ lex odd_even.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ gcc lex.yy.c -o odd_even -lfl
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ ./odd_even
Enter numbers (Ctrl+D to end):
80
80 is Even
63
63 is Odd
```

3. Aim: To write a program that identifies and counts vowels in each character or string.

Program:

```
%{
#include <stdio.h>
int vowels=0;
int cons=0;
%}
%%
[aeiouAEIOU] { vowels++; }
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { cons++; }
%%
int yywrap()
{
    return 1;
}
int main()
{
    printf("Enter the string.. at end press ^d\n");
    yylex();
    printf("No of vowels=%d\nNo of consonants=%d\n", vowels, cons);
    return 0;
}
```

Output:

```
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano vowel_check.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ lex vowel_check.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ gcc lex.yy.c -o vowel_check -lfl
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ ./vowel_check
Enter text (Ctrl+D to end):
anindita
a is a vowel
n is NOT a vowel
i is a vowel
n is NOT a vowel
d is NOT a vowel
i is a vowel
t is NOT a vowel
a is a vowel
```

4. Aim: To write a program that determines whether a given number is positive or negative.

Program:

```
  GNU nano 2.9.3                                    pos_neg.l

%{
#include <stdio.h>
int pos = 0, neg = 0;
%}

%%
[-][0-9]+     { neg++; }
[0-9]+        { pos++; }
.|\n          ;
%%

int yywrap() { return 1; }

int main() {
    printf("Enter numbers (Ctrl+D to end):\n");
    yylex();
    printf("Positive numbers: %d\n", pos);
    printf("Negative numbers: %d\n", neg);
    return 0;
}
```

Output:

```
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano pos_neg.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ lex pos_neg.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ gcc lex.yy.c -o pos_neg -lfl
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ ./pos_neg
Enter numbers (Ctrl+D to end):
3
-4
Positive numbers: 1
Negative numbers: 1
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$
```

5. Aim: To write a program that calculates the sum of the digits of a given number.

Program:

```
GNU nano 2.9.3                                    sum_digit.l

%{
#include <stdio.h>
int sum = 0;
%}

%%
[0-9] { sum += atoi(yytext); }
.|\n   ;
%%

int yywrap() { return 1; }

int main() {
    printf("Enter text (Ctrl+D to end):\n");
    yylex();
    printf("Sum of digits: %d\n", sum);
    return 0;
}
```

Output:

```
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ nano sum_digit.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ lex sum_digit.l
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ gcc lex.yy.c -o sum_digit -lfl
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$ ./sum_digit
Enter text (Ctrl+D to end):
123
Sum of digits: 6
asecomputerlab@ase-computerlab:~/compiler_lab1/programs$
```

Results: The programs for Implementation of Lexical Analyzer using Lex Tools has been successfully executed.

Lab Exercise -02

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

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:~$
```

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++;
            }
```

```
            }

            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:~$
```

<u>**Results**</u>: The program to implement left factoring and left recursion has been successfully executed.

Lab Exercises - 03

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

Code:

```
#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 '$'
```

```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--) {
```

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.

Lab Exercises -04

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

Code:

```
%{
#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;
}
```

Output:

```
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
```

Results: The program in YACC for parser generation has been executed successfully

Lab Exercise- 05

Aim: To Implement Symbol Table

Code:

```c
#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");
```

```c
    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:

```
Input the expression ending with $ sign: a+b=c$

Given Expression: a+b=c

Symbol Table display
Symbol   Address         Type
a        0x564c531d6ac0          identifier
+        0x564c531d6ae0          operator
b        0x564c531d6b00          identifier
=        0x564c531d6b20          operator
c        0x564c531d6b40          identifier
```

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

Lab Exercise- 06

Aim: To Implement Intermediate Code generation

Code:

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

int i = 1, j = 0, no = 0, tmpch = 90; // Temporary variables for indexing and character generation
char str[100], left[15], right[15]; // Input expression, left and right operands

struct exp {
    int pos; // Position of operator
    char op; // Operator
} k[15];

void findopr();
void explore();
void fleft(int);
void fright(int);

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

    findopr();  // Find all operators
    explore();  // Generate intermediate code

    return 0;
}
```

```c
void findopr() {
    // Searching for operators in the input string and storing their positions
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == ':') {
            k[j].pos = i;
            k[j++].op = ':';
        }
    }
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '/') {
            k[j].pos = i;
            k[j++].op = '/';
        }
    }
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '*') {
            k[j].pos = i;
            k[j++].op = '*';
        }
    }
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '+') {
            k[j].pos = i;
            k[j++].op = '+';
        }
    }
    for (i = 0; str[i] != '\0'; i++) {
        if (str[i] == '-') {
            k[j].pos = i;
            k[j++].op = '-';
        }
    }
}
```

```c
void explore() {
    i = 0; // Start with the first operator
    while (k[i].op != '\0') { // Loop through all the operators
        fleft(k[i].pos);   // Find left operand
        fright(k[i].pos);  // Find right operand
        str[k[i].pos] = tmpch--;  // Assign temporary variable to the operator position

        // Print intermediate code in the form of: `temp := left op right`
        printf("\t%c := %s %c %s\n", str[k[i].pos], left, k[i].op, right);

        i++; // Move to the next operator
    }

    // Handle the last remaining operation if any
    fright(-1);
    if (no == 0) {
        fleft(strlen(str));
        printf("\t%s := %s\n", right, left);
        exit(0);
    }

    printf("\t%s := %c\n", right, str[k[--i].pos]);
}
```

```c
// Function to find the left operand for an operator
void fleft(int x) {
    int w = 0, flag = 0;
    x--; // Move left from operator position
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '=' && str[x] != '\0' && str[x] != '-' && str[x] != '/' && str[x] != ':') {
        if (str[x] != '$' && flag == 0) {
            left[w++] = str[x]; // Add character to left operand
            left[w] = '\0';
            str[x] = '$'; // Mark as visited
            flag = 1;
        }
        x--; // Move further left
    }
}

// Function to find the right operand for an operator
void fright(int x) {
    int w = 0, flag = 0;
    x++; // Move right from operator position
    while (x != -1 && str[x] != '+' && str[x] != '*' && str[x] != '\0' && str[x] != '=' && str[x] != ':' && str[x] != '-' && str[x] != '/') {
        if (str[x] != '$' && flag == 0) {
            right[w++] = str[x]; // Add character to right operand
            right[w] = '\0';
            str[x] = '$'; // Mark as visited
            flag = 1;
        }
        x++; // Move further right
    }
}
```

Output:

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

```
                INTERMEDIATE CODE GENERATION

Enter the Expression : a+b*c-d/e$
The intermediate code:
        Z := d / e
        Y := b * c
        X := a + Y
        W := X - Z
        W := X
```

executed successfully

Lab Exercise- 07

Aim: To implementation of Code Optimization Techniques

Code:

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

struct op {
    char l;      // Left-hand side (e.g., a = ...)
    char r[20];  // Right-hand side (e.g., ... = b + c)
} op[10], pr[10];

int main() {
    int i, j, n, z = 0, k, m, q, a;
    char temp, t;
    char *p, *l;
    char *tem;

    printf("Enter the number of expressions: ");
    scanf("%d", &n);

    // Input expressions
    for (i = 0; i < n; i++) {
        printf("Left: ");
        scanf(" %c", &op[i].l); // Space before %c to ignore newline
        printf("Right: ");
        scanf(" %s", op[i].r);
    }

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

```c
    // DEAD CODE ELIMINATION
    for (i = 0; i < n - 1; i++) {
        temp = op[i].l;
        for (j = 0; j < n; j++) {
            p = strchr(op[j].r, temp); // Check if temp is used in any expression
            if (p) {
                pr[z].l = op[i].l;
                strcpy(pr[z].r, op[i].r);
                z++;
                break;
            }
        }
    }

    // Add the last expression (assumed as output-related)
    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;
        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;
                        pr[i].r[a] = pr[m].l;
                    }
                }
            }
        }
    }

    printf("\nAfter Eliminating Common Subexpressions:\n");
    for (i = 0; i < z; i++) {
        printf("%c = %s\n", pr[i].l, pr[i].r);
    }

    // REMOVE DUPLICATE ASSIGNMENTS (e.g., a=b+c and again a=b+c)
    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[j].l = '\0'; // Mark for removal
            }
        }
    }
```

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

    return 0;
}
```

Output:

```
Enter the number of expressions: 4
Left: a
Right: b+c
Left: d
Right: a+e
Left: f
Right: b+c
Left: g
Right: f+h
```

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

After Dead Code Elimination:
a = b+c
f = b+c
g = f+h

After Eliminating Common Subexpressions:
a = b+c
a = b+c
g = a+h

Optimized Code:
a = b+c
g = a+h
```

Result: Thus, the program to implement code optimization has been executed successfully

Lab Exercise- 08

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

Code:

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

int label[20]; // For storing label numbers
int no = 0;    // Label counter

// Function declaration
int check_label(int k);

int main() {
    FILE *fp1, *fp2;
    char fname[100];
    char op[20], operand1[20], operand2[20], result[20];
    char ch;
    int i = 0;

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

    fp1 = fopen(fname, "r");
    fp2 = fopen("target.txt", "w");

    if (fp1 == NULL || fp2 == NULL) {
        printf("\nError opening file.\n");
        exit(1);
    }
```

```c
    // Read the intermediate code line by line
    while (fscanf(fp1, "%s", op) != EOF) {
        i++;

        if (check_label(i)) {
            fprintf(fp2, "\nLABEL#%d:\n", i);
        }

        if (strcmp(op, "print") == 0) {
            fscanf(fp1, "%s", result);
            fprintf(fp2, "\tOUT %s\n", result);
        }
        else if (strcmp(op, "goto") == 0) {
            fscanf(fp1, "%s %s", operand1, operand2);
            fprintf(fp2, "\tJMP %s, LABEL#%s\n", operand1, operand2);
            label[no++] = atoi(operand2);
        }
        else if (strcmp(op, "[]=") == 0) {
            fscanf(fp1, "%s %s %s", operand1, operand2, result);
            fprintf(fp2, "\tSTORE %s[%s], %s\n", operand1, operand2, result);
        }
        else if (strcmp(op, "uminus") == 0) {
            fscanf(fp1, "%s %s", operand1, result);
            fprintf(fp2, "\tLOAD -%s, R1\n", operand1);
            fprintf(fp2, "\tSTORE R1, %s\n", result);
        }
```

```c
        else {
            // Handle arithmetic and relational operations using first character
            switch (op[0]) {
                case '*':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s, R1\n", operand2);
                    fprintf(fp2, "\tMUL R1, R0\n");
                    fprintf(fp2, "\tSTORE R0, %s\n", result);
                    break;
                case '+':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s, R1\n", operand2);
                    fprintf(fp2, "\tADD R1, R0\n");
                    fprintf(fp2, "\tSTORE R0, %s\n", result);
                    break;
                case '-':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s, R1\n", operand2);
                    fprintf(fp2, "\tSUB R1, R0\n");
                    fprintf(fp2, "\tSTORE R0, %s\n", result);
                    break;
                case '/':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s, R1\n", operand2);
                    fprintf(fp2, "\tDIV R1, R0\n");
                    fprintf(fp2, "\tSTORE R0, %s\n", result);
                    break;
                case '%':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tLOAD %s, R1\n", operand2);
                    fprintf(fp2, "\tMOD R1, R0\n");
                    fprintf(fp2, "\tSTORE R0, %s\n", result);
                    break;
```

```c
                    break;
                case '=':
                    fscanf(fp1, "%s %s", operand1, result);
                    fprintf(fp2, "\tSTORE %s, %s\n", operand1, result);
                    break;
                case '>':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tJGT R0, %s, LABEL#%s\n", operand2, result);
                    label[no++] = atoi(result);
                    break;
                case '<':
                    fscanf(fp1, "%s %s %s", operand1, operand2, result);
                    fprintf(fp2, "\tLOAD %s, R0\n", operand1);
                    fprintf(fp2, "\tJLT R0, %s, LABEL#%s\n", operand2, result);
                    label[no++] = atoi(result);
                    break;
                default:
                    // Unsupported operation
                    fprintf(fp2, "\t; Unknown operation: %s\n", op);
            }
        }
    }

    fclose(fp1);
    fclose(fp2);
```

Input.txt

```
= a t1
= b t2
+ t1 t2 t3
= t3 c
print c
```

Output:

```
Enter filename of the intermediate code: input.txt

Generated Target Code:

        STORE a, t1
        STORE b, t2
        LOAD t1, R0
        LOAD t2, R1
        ADD R1, R0
        STORE R0, t3
        STORE t3, c
        OUT c
```

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