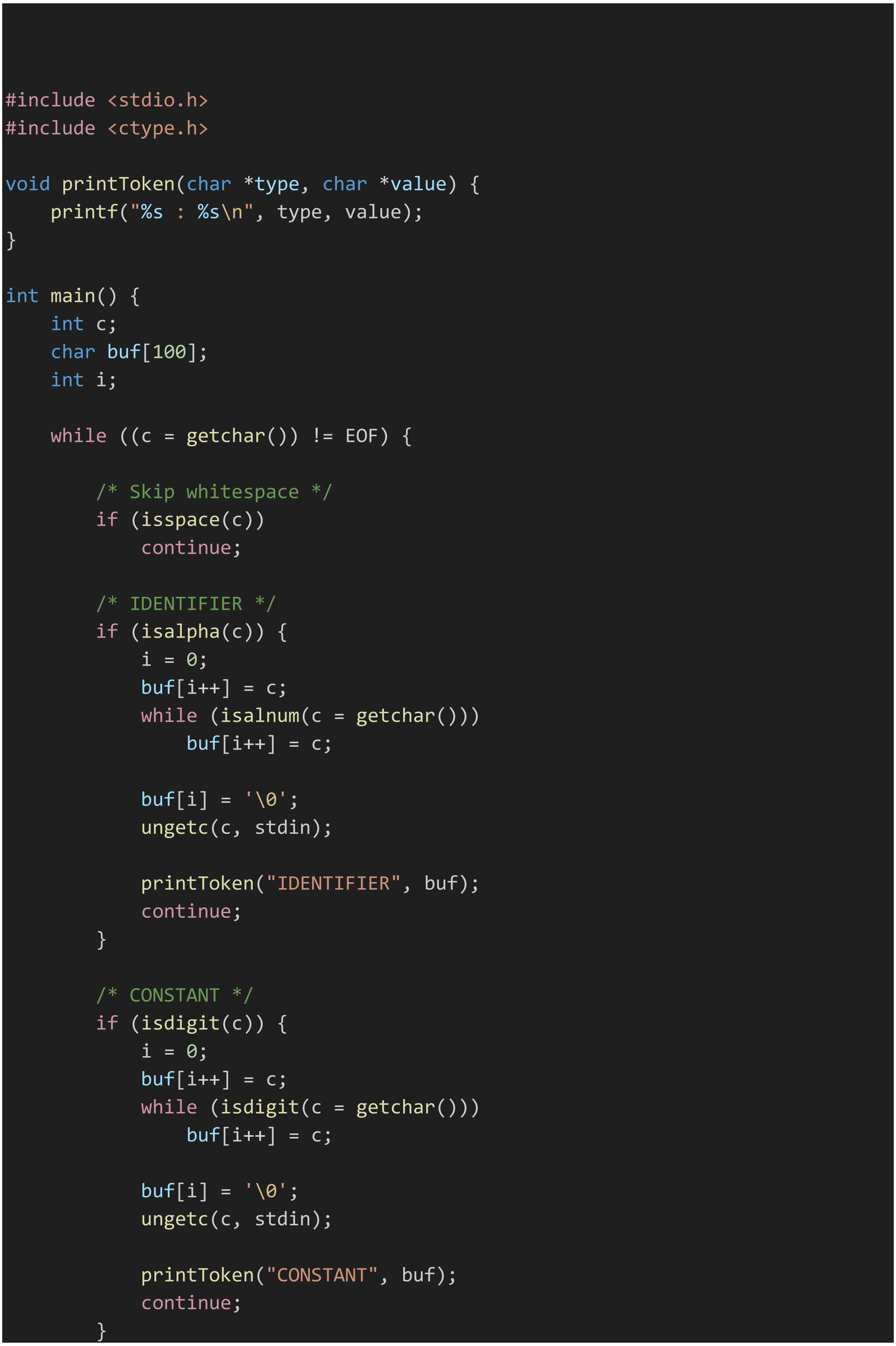
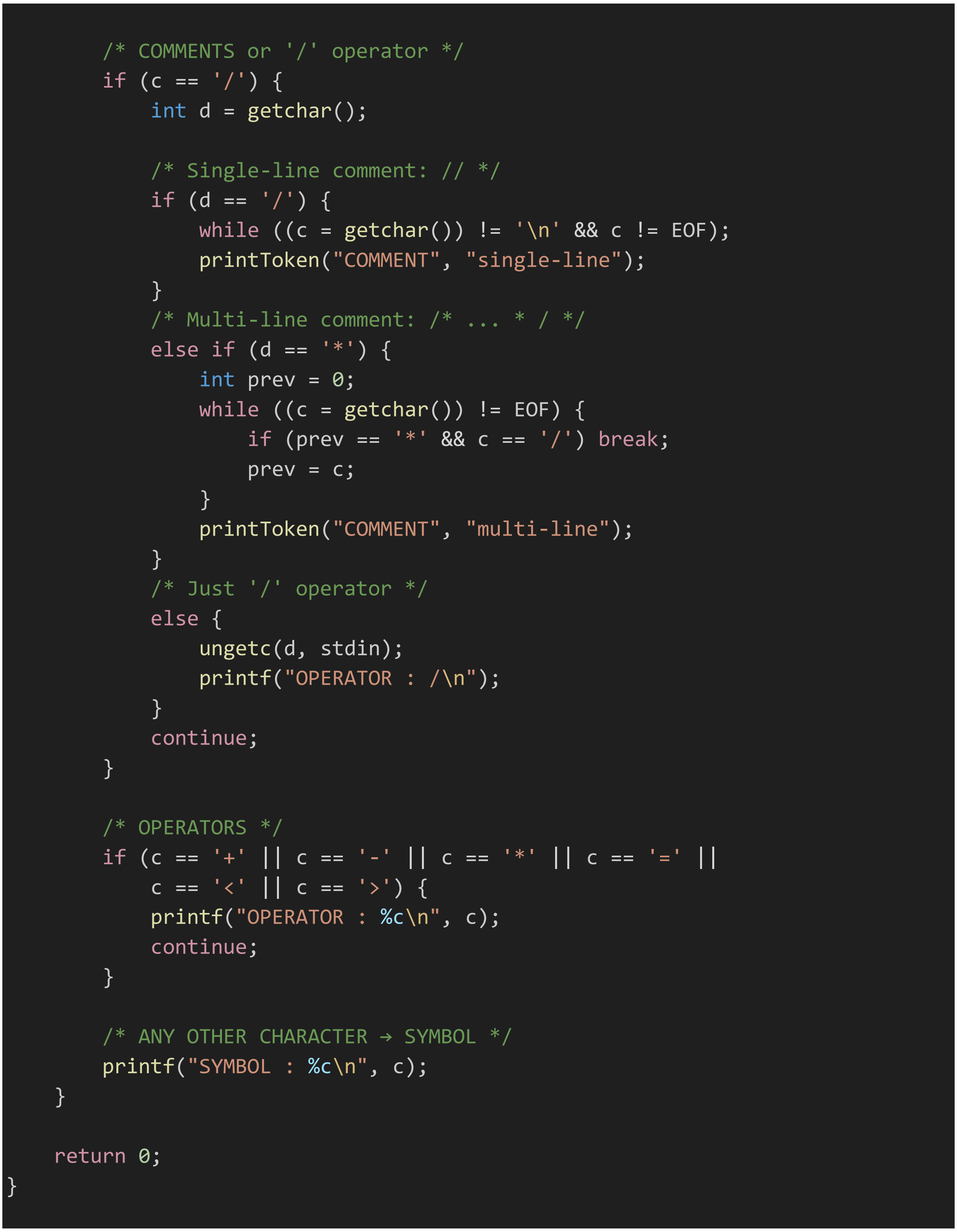


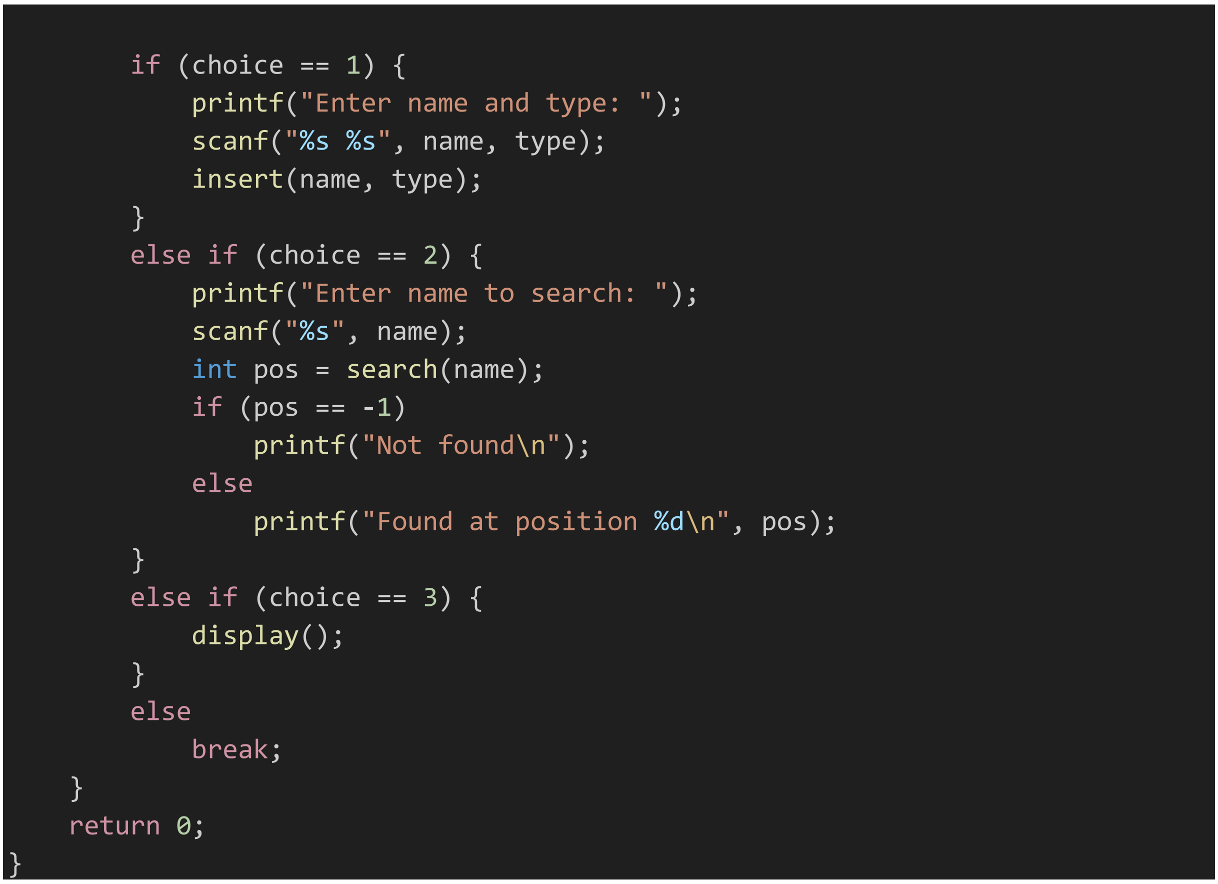
```c
#include <stdio.h>
#include <ctype.h>

void printToken(char *type, char *value) {
    printf("%s : %s\n", type, value);
}

int main() {
    int c;
    char buf[100];
    int i;

    while ((c = getchar()) != EOF) {

        /* Skip whitespace */
        if (isspace(c))
            continue;

        /* IDENTIFIER */
        if (isalpha(c)) {
            i = 0;
            buf[i++] = c;
            while (isalnum(c = getchar()))
                buf[i++] = c;

            buf[i] = '\0';
            ungetc(c, stdin);

            printToken("IDENTIFIER", buf);
            continue;
        }

        /* CONSTANT */
        if (isdigit(c)) {
            i = 0;
            buf[i++] = c;
            while (isdigit(c = getchar()))
                buf[i++] = c;

            buf[i] = '\0';
            ungetc(c, stdin);

            printToken("CONSTANT", buf);
            continue;
        }
```



```
        /* COMMENTS or '/' operator */
        if (c == '/') {
            int d = getchar();

            /* Single-line comment: // */
            if (d == '/') {
                while ((c = getchar()) != '\n' && c != EOF);
                printToken("COMMENT", "single-line");
            }
            /* Multi-line comment: /* ... * / */
            else if (d == '*') {
                int prev = 0;
                while ((c = getchar()) != EOF) {
                    if (prev == '*' && c == '/') break;
                    prev = c;
                }
                printToken("COMMENT", "multi-line");
            }
            /* Just '/' operator */
            else {
                ungetc(d, stdin);
                printf("OPERATOR : /\n");
            }
            continue;
        }

        /* OPERATORS */
        if (c == '+' || c == '-' || c == '*' || c == '=' ||
            c == '<' || c == '>') {
            printf("OPERATOR : %c\n", c);
            continue;
        }

        /* ANY OTHER CHARACTER → SYMBOL */
        printf("SYMBOL : %c\n", c);
    }

    return 0;
}
```

Symbol table

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

#define MAX 50

struct Symbol {
    char name[30];
    char type[10];
} table[MAX];

int count = 0;

/* Insert a new symbol */
void insert(char name[], char type[]) {
    strcpy(table[count].name, name);
    strcpy(table[count].type, type);
    count++;
}

/* Search for a symbol */
int search(char name[]) {
    for (int i = 0; i < count; i++) {
        if (strcmp(table[i].name, name) == 0)
            return i;
    }
    return -1;
}

/* Display table */
void display() {
    printf("\nSymbol Table:\n");
4

    for (int i = 0; i < count; i++)
        printf("%s\t%s\n", table[i].name, table[i].type);
}

int main() {
    int choice;
    char name[30], type[10];

    while (1) {
        printf("\n1.Insert  2.Search  3.Display  4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
```



```c
        if (choice == 1) {
            printf("Enter name and type: ");
            scanf("%s %s", name, type);
            insert(name, type);
        }
        else if (choice == 2) {
            printf("Enter name to search: ");
            scanf("%s", name);
            int pos = search(name);
            if (pos == -1)
                printf("Not found\n");
            else
                printf("Found at position %d\n", pos);
        }
        else if (choice == 3) {
            display();
        }
        else
            break;
    }
    return 0;
}
```

Identifier and flow keyword

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

int main() {
    int i, flag = 0;
    char str[50];

    printf("Enter string: ");
    scanf("%s", str);

    // Check for control-flow keywords
    if( strcmp(str,"if")==0 || strcmp(str,"else")==0 || strcmp(str,"do")==0 ||
        strcmp(str,"for")==0 || strcmp(str,"break")==0 ||
strcmp(str,"while")==0 ||
        strcmp(str,"switch")==0 || strcmp(str,"case")==0 ||
strcmp(str,"default")==0 )
    {
        printf("Keyword of control flow statements\n");
        return 0;
    }
    // Check for identifier
    if (isalpha(str[0]) || str[0] == '_') {
        flag = 0;
        for (i = 1; i < strlen(str); i++) {
            if (!(isalnum(str[i]) || str[i] == '_')) {
                flag = 1;    // invalid character found
6


                break;
            }
        }

        if (flag == 0)
            printf("Identifier\n");
        else
            printf("Not a keyword or identifier\n");
    }
    else {
        printf("Not a keyword or identifier\n");
    }

    return 0;
}
```

Type checking

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

struct Symbol {
    char name[20];
    char type[10];
} table[] = {
    {"a", "int"},
    {"b", "int"},
    {"c", "float"},
    {"d", "float"}
};

char* getType(char var[]) {


    for (int i = 0; i < 4; i++)
        if (strcmp(table[i].name, var) == 0)
            return table[i].type;
    return "unknown";
}

int main() {
    char op1[20], op2[20];

    printf("Enter two operands: ");
    scanf("%s %s", op1, op2);

    char *t1 = getType(op1);
    char *t2 = getType(op2);

    if (strcmp(t1, t2) == 0)
        printf("Type Check Passed: Both are %s\n", t1);
    else
        printf("Type Error: %s is %s, but %s is %s\n", op1, t1, op2, t2);

    return 0;
}
```

Static

```c
#include <stdio.h>

void incrementCounter() {
static int counter = 0;
counter++;
printf("Static counter value = %d  (addr: %p)\n", counter, (void *)&counter);
}

int main(void) {

printf("Calling incrementCounter() first time...\n");
incrementCounter();

printf("\nCallingincrementCounter() second time...\n");

incrementCounter();

printf("\nCallingincrementCounter() third time...\n");
incrementCounter();

printf("\nProgram finished.\n");

return 0;
}
```

Stack

```c
#include <stdio.h>

void incrementStackCounter() {
    int counter = 0;   // STACK VARIABLE (automatic)
                       // Created & destroyed on each call
    counter++;         // Always becomes 1
    printf("Stack counter value = %d  (addr: %p)\n", counter, (void
*)&counter);
}

int main(void) {

    printf("Calling incrementStackCounter() first time...\n");
    incrementStackCounter();

    printf("\nCalling incrementStackCounter() second time...\n");
    incrementStackCounter();
    printf("\nCalling incrementStackCounter() third time...\n");
    incrementStackCounter();

    printf("\nProgram finished.\n");

    return 0;
}
```

Heap

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

void incrementHeapValue(int *ptr) {
    (*ptr)++;   // Increase the value stored in heap
    printf("Heap value = %d  (addr: %p)\n", *ptr, (void *)ptr);
}

int main() {
    // Allocate 1 integer on the heap
    int *heapVar = (int *)malloc(sizeof(int));

    if (heapVar == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    *heapVar = 0;   // initialize heap value

    printf("Calling incrementHeapValue() first time...\n");
    incrementHeapValue(heapVar);

    printf("\nCalling incrementHeapValue() second time...\n");
    incrementHeapValue(heapVar);

    printf("\nCalling incrementHeapValue() third time...\n");
    incrementHeapValue(heapVar);

    // Free heap memory
    free(heapVar);

    printf("\nMemory freed. Program finished.\n");

    return 0;
}
```

Predictive parser

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

char st[50];
int top = -1;

void push(char c){ st[++top] = c; }
void pop(){ top--; }

int main(){
    char in[50];
    scanf("%s", in);

    int p = 0;

    push('$');
    push('E');

    while(top >= 0){
        char X = st[top];
        char a = in[p];

        // Terminal match
        if(X == a){
            pop();
            p++;
            if(a == '$'){
                printf("ACCEPT\n");
                return 0;
            }
        }

        // E → T R
        else if(X == 'E'){
            pop();
            push('R');
            push('T');
        }

        // R → + T R | ε
        else if(X == 'R'){
            if(a == '+'){
                pop();
                push('R');
                push('T');
                push('+');
            } else {
                pop();  // epsilon
            }
        }

        // T → F P
        else if(X == 'T'){
            pop();
            push('P');
            push('F');
        }

        // P → * F P | ε
        else if(X == 'P'){
            if(a == '*'){
                pop();
                push('P');
                push('F');
                push('*');
            } else {
                pop(); // epsilon
            }
        }

        // F → i | (E)
        else if(X == 'F'){
            pop();
            if(a == 'i'){
                push('i');
            }
            else if(a == '('){
                push(')');
                push('E');
                push('(');
            }
            else {
                printf("ERROR\n");
                return 0;
            }
        }

        // Nothing matches
        else{
            printf("ERROR\n");
            return 0;
        }
    }
}
```

.Write a C program to construct Recursive Descent parser for the following grammar

E→TR

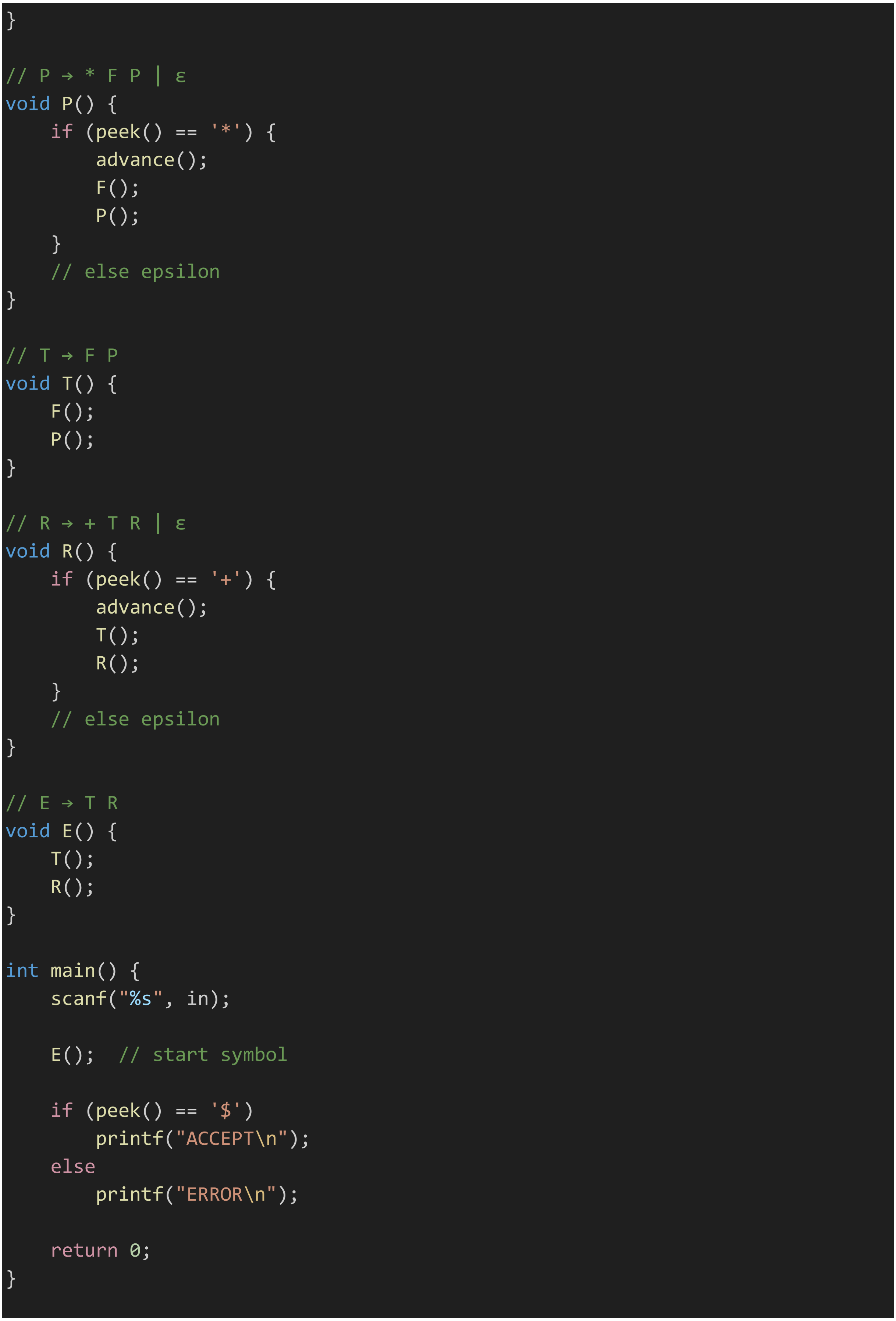
R→+TR/∈

T→FP

T→*FP/∈

F→a/(E)

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

char in[100];
int p = 0;

char peek() { return in[p]; }
void advance() { p++; }
void error() {
    printf("ERROR\n");
    exit(0);
}

void E();
void R();
void T();
void P();
void F();

// F → i | (E)
void F() {
    if (peek() == 'i') {
        advance();
    }
    else if (peek() == '(') {
        advance();      // '('
        E();            // parse inside expression
        if (peek() == ')')
            advance(); // ')'
        else
            error();
    }
    else {
        error();
    }
```



```c
}

// P → * F P | ε
void P() {
    if (peek() == '*') {
        advance();
        F();
        P();
    }
    // else epsilon
}

// T → F P
void T() {
    F();
    P();
}

// R → + T R | ε
void R() {
    if (peek() == '+') {
        advance();
        T();
        R();
    }
    // else epsilon
}

// E → T R
void E() {
    T();
    R();
}

int main() {
    scanf("%s", in);

    E();  // start symbol

    if (peek() == '$')
        printf("ACCEPT\n");
    else
        printf("ERROR\n");

    return 0;
}
```

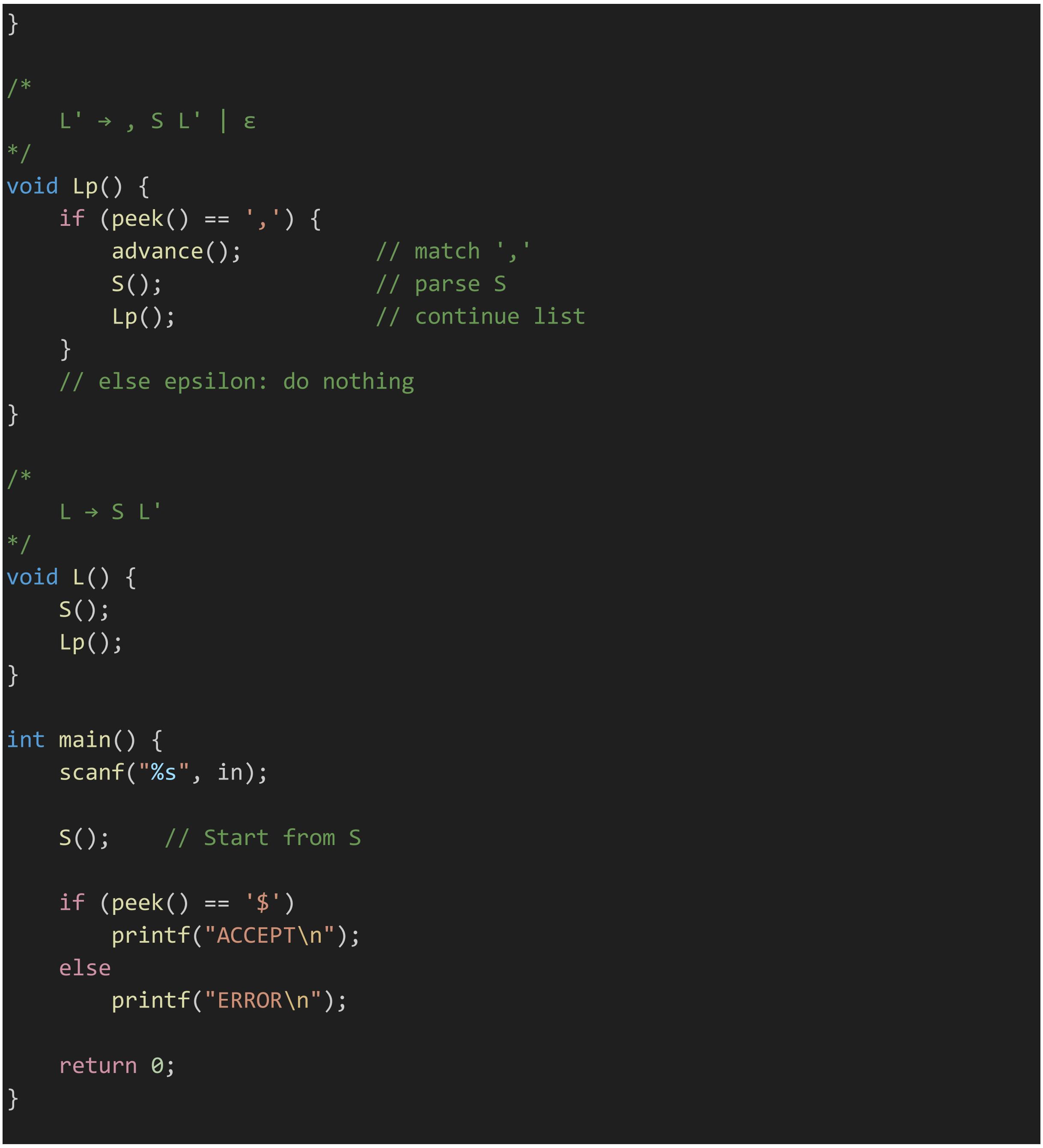
write recursive descent parser for the grammar S->(L) S->a L->L,S L->S

S → (L) | a

L → S L'

L' → , S L' | ε

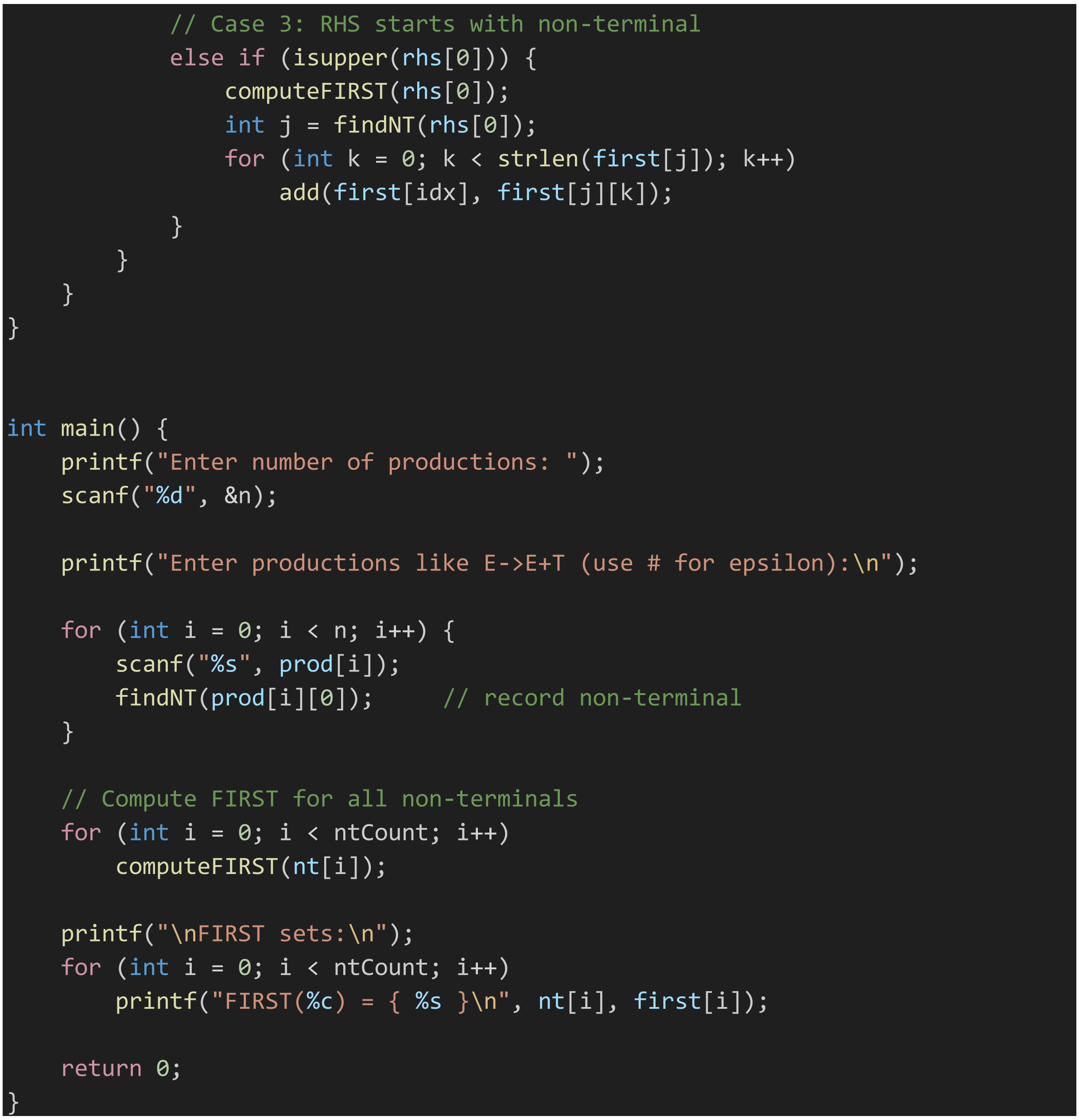
```
#include <stdio.h>
#include <stdlib.h>

char in[100];
int p = 0;

char peek() {
    return in[p];
}

void advance() {
    p++;
}

void error() {
    printf("ERROR\n");
    exit(0);
}

void S();
void L();
void Lp();

/*
    S → a | (L)
*/
void S() {
    if (peek() == 'a') {
        advance();          // match a
    }
    else if (peek() == '(') {
        advance();          // match '('
        L();                // parse list
        if (peek() == ')')
            advance();      // match ')'
        else
            error();
    }
    else {
        error();
    }
```



```c
}

/*
    L' → , S L' | ε
*/
void Lp() {
    if (peek() == ',') {
        advance();          // match ','
        S();                // parse S
        Lp();               // continue list
    }
    // else epsilon: do nothing
}

/*
    L → S L'
*/
void L() {
    S();
    Lp();
}

int main() {
    scanf("%s", in);

    S();    // Start from S

    if (peek() == '$')
        printf("ACCEPT\n");
    else
        printf("ERROR\n");

    return 0;
}
```

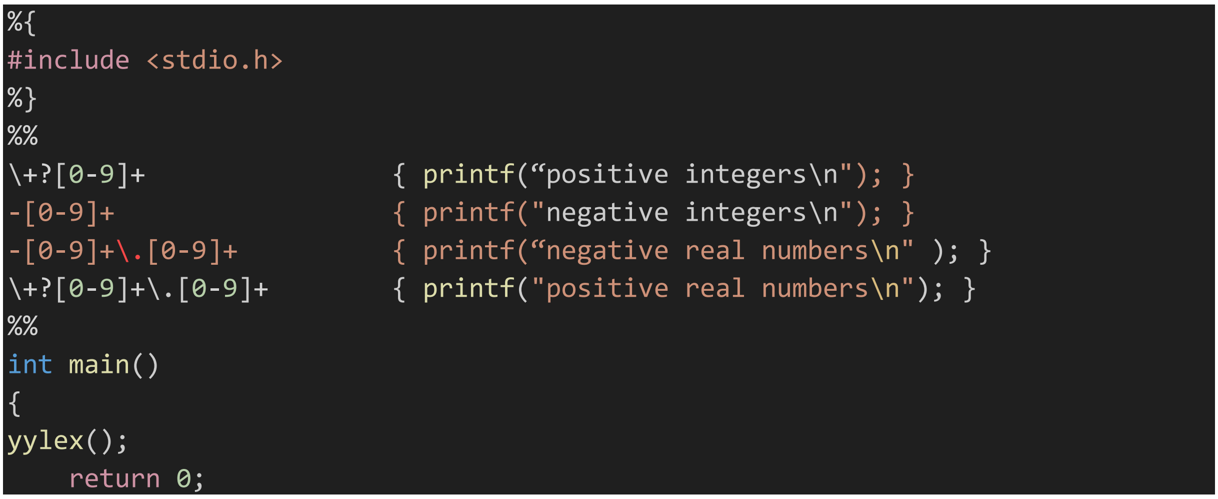
First of grammar

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

char prod[20][20];      // productions
char first[20][20];     // FIRST sets
char nt[20];            // non-terminals list
int n, ntCount = 0;     // number of productions, number of NTs
int done[256] = {0};    // memoization flag

int findNT(char c) {
23

    for (int i = 0; i < ntCount; i++)
        if (nt[i] == c)
            return i;
    nt[ntCount] = c;
    first[ntCount][0] = '\0';
    return ntCount++;
}

void add(char set[], char c) {
    if (!strchr(set, c)) {
        int l = strlen(set);
        set[l] = c;
        set[l+1] = '\0';
    }
}

// Compute FIRST for non-terminal X
void computeFIRST(char X) {
    int idx = findNT(X);
    if (done[X]) return;   // avoid infinite recursion on left-recursion
    done[X] = 1;

    for (int i = 0; i < n; i++) {
        if (prod[i][0] == X) {
            char *rhs = strchr(prod[i], '>') + 1;

            // Case 1: RHS starts with terminal or symbol like '(' '+'
            if (!isupper(rhs[0]) && rhs[0] != '#') {
                add(first[idx], rhs[0]);
            }
            // Case 2: RHS starts with epsilon
            else if (rhs[0] == '#') {
                add(first[idx], '#');
            }
```



```
            // Case 3: RHS starts with non-terminal
            else if (isupper(rhs[0])) {
                computeFIRST(rhs[0]);
                int j = findNT(rhs[0]);
                for (int k = 0; k < strlen(first[j]); k++)
                    add(first[idx], first[j][k]);
            }
        }
    }
}


int main() {
    printf("Enter number of productions: ");
    scanf("%d", &n);

    printf("Enter productions like E->E+T (use # for epsilon):\n");

    for (int i = 0; i < n; i++) {
        scanf("%s", prod[i]);
        findNT(prod[i][0]);     // record non-terminal
    }

    // Compute FIRST for all non-terminals
    for (int i = 0; i < ntCount; i++)
        computeFIRST(nt[i]);

    printf("\nFIRST sets:\n");
    for (int i = 0; i < ntCount; i++)
        printf("FIRST(%c) = { %s }\n", nt[i], first[i]);

    return 0;
}
```

Lex programs:



```
%{
#include <stdio.h>
%}
%%
\+?[0-9]+                { printf(“positive integers\n"); }
-[0-9]+                  { printf("negative integers\n"); }
-[0-9]+\.[0-9]+          { printf(“negative real numbers\n" ); }
\+?[0-9]+\.[0-9]+        { printf("positive real numbers\n"); }
%%
int main()
{
yylex();
    return 0;
}
```

7.Write a Lex specification for converting real numbers to integers.

```
 %{

int i, j;
#include <stdio.h>
#include <stdlib.h>
%}
%%
[0-9]*\.[0-9]+  {
                    for (i = 0; i<yyleng; i++)
                    {
                        if (yytext[i] == '.')
                        {
                            for (j = 0; j <= i - 1; j++)
                                    printf("%c", yytext[j]);
                            break;
                        }
                    }
                }
%%
int main(void)
{
yylex();
    return 0;
}
```

.Write a Lex specification to print the number of days in a month using a procedure

```
%{

#include <stdio.h>
int year;
void leap(void);        /* prototype */
%}

%%

jan|mar|may|july|aug|oct|dec{ printf("31 days"); }
april|june|sep|nov{ printf("30 days"); }
```

```
feb                             { leap(); }
[a-zA-Z]*                       { printf("invalid"); }

%%

main()
{
yylex();
}

void leap(void)
{
    printf("enter year: ");
    scanf("%d", &year);

    if ((year % 400 == 0) || (year % 4 == 0 && year % 100 != 0))
        printf("29 days");
    else
        printf("28 days");
}
```

. Write a Lex specification to retrieve comments.

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

%%

"//".*              { printf("%s\n", yytext + 2); }

"/*"([^*]|\*+[^/])*"*/" {
                          int i;
                         for(i = 2; yytext[i] && !(yytext[i]=='*' &&
yytext[i+1]=='/'); i++)
                               putchar(yytext[i]);
                           printf("\n");
                        }

.|\n               ;     /* ignore everything else */

%%

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

Write a Lex specification to design a lexical analyzer that recognizes identifiers and keywords of flow control statements of C language

```
%{
 #include<stdio.h>
%}
%%
If|else|while|do|switch|case|break|for|default {printf("Keyword");}
IF|ELSE|WHILE|DO|SWITCH|CASE|BREAK|FOR|DEFAULT  {printf("Keyword");}
[A-Z a-z]+[a-z A-Z 0-9 _]*        {printf("identifier");}
%%
int main()
{

yylex();
return 0;
}
```

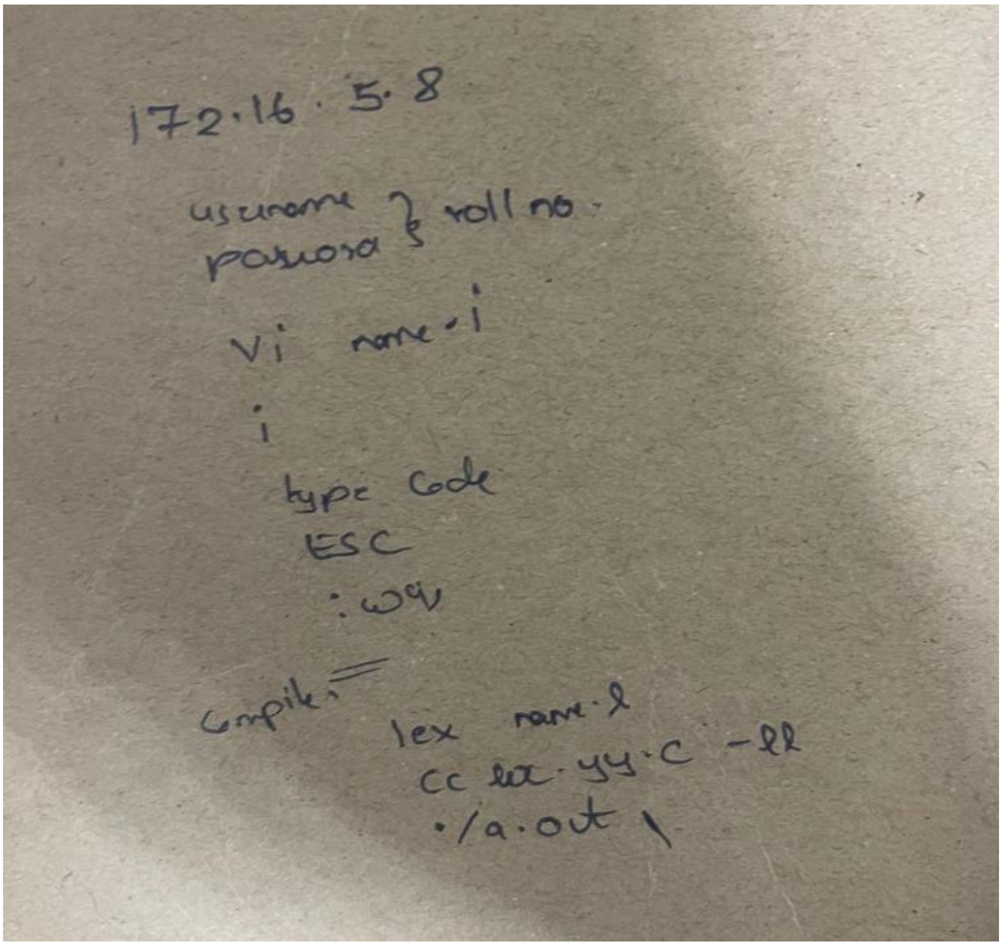
Implementation of lexical analyzer using lex tool.

```
%{


#include <stdio.h>
%}

%%

[0-9]+                      { printf("NUMBER\t%s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*      { printf("IDENTIFIER\t%s\n", yytext); }
[+\-*/=]                    { printf("OPERATOR\t%s\n", yytext); }
```

```
[ \t\n]                       ;       /* ignore spaces, tabs, newline */

%%

int main() {
    yylex();
    return 0;
}
```

Write a lex program to count the number of words and number of lines in a given file or program

```
%{
int wc=0, lc=0;
%}

%%

\n        { lc++; }
[ \t]+    ;
[^ \t\n]+ { wc++; }

%%

int main() {
    yylex();
    printf("Lines=%d\nWords=%d\n", lc, wc);
    return 0;
}
```



172.16.5.8

username } roll no.
password }

```
vi  name.i

i
type code
ESC
:wq
```

Compile:

```
lex  name.l
cc lex.yy.c  -ll
./a.out
```