BCSE307P

Compiler Design Lab

Lab Assignment 3

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

Name – Ishan Kapoor

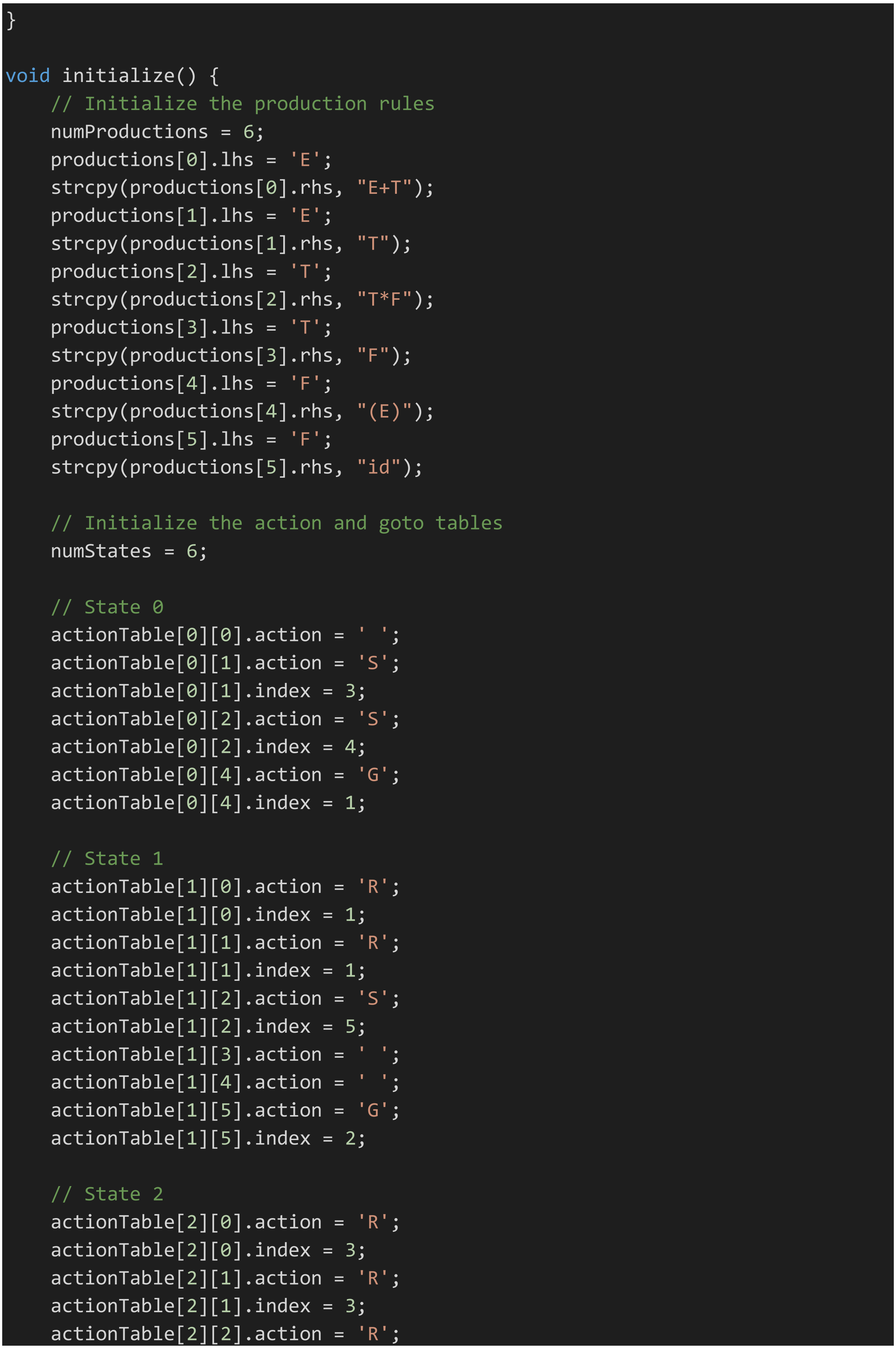
Registration Number – 21BCE5882

Submitted to – Prof. S. Srisakthi

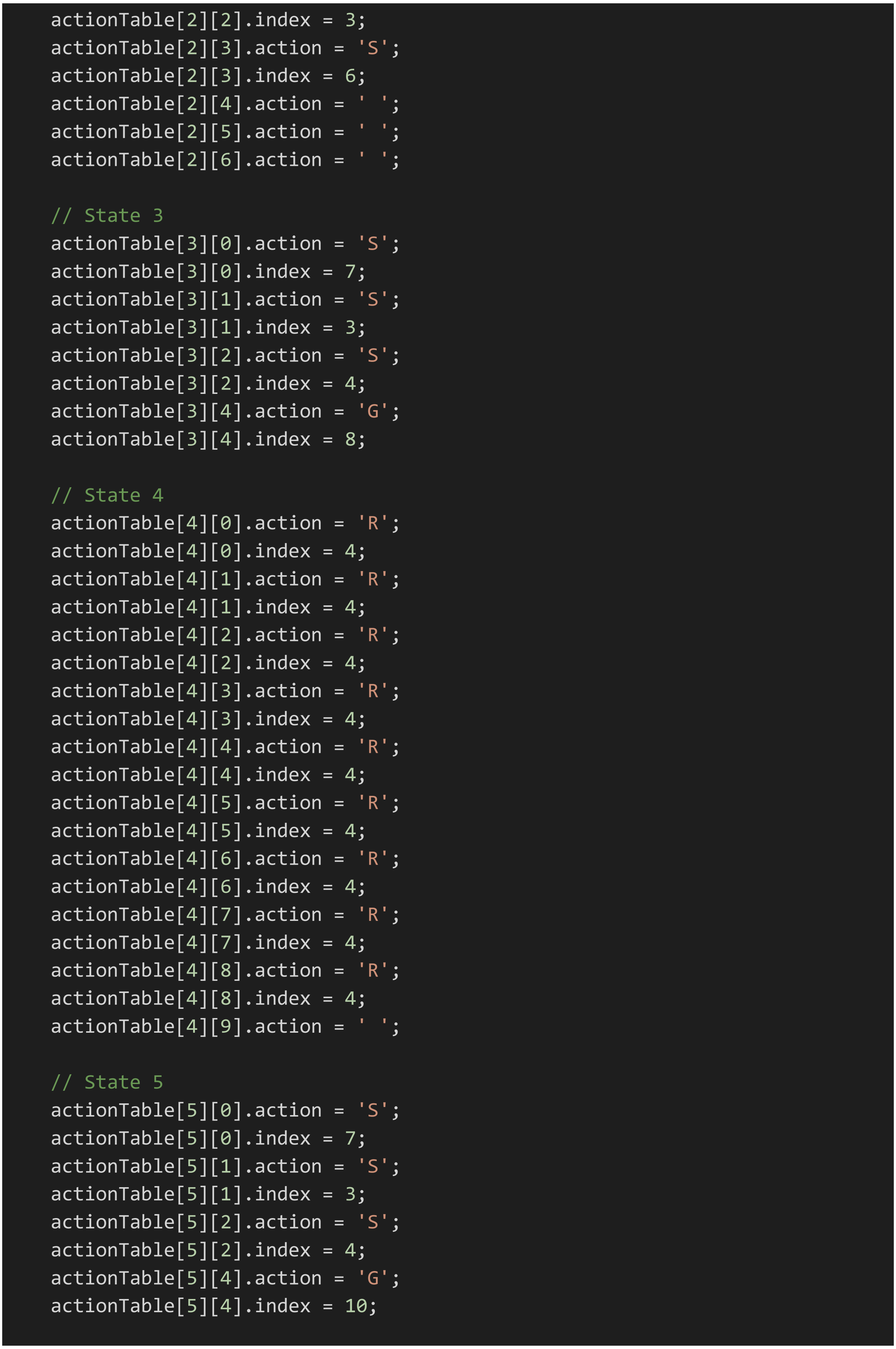
1. A C program for LALR parsing

Code:

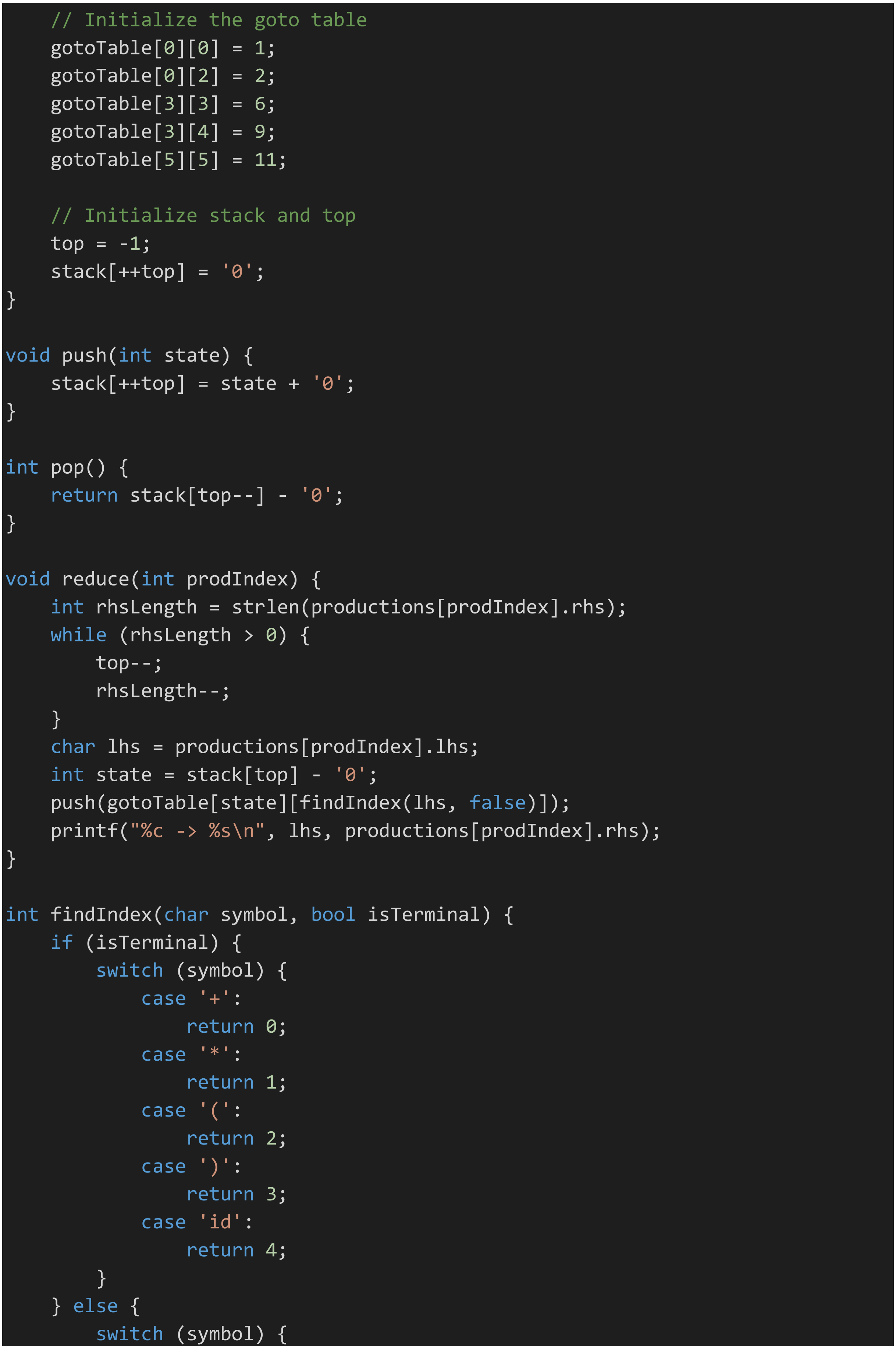
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX_STACK_SIZE 100
#define MAX_INPUT_SIZE 100

// Production structure
typedef struct {
    char lhs; // Left-hand side of the production
    char rhs[10]; // Right-hand side of the production (assuming maximum 10
symbols)
} Production;

// Action table entry
typedef struct {
    char action; // Action to be performed (S, R, or A)
    int index; // Index of the state or production
} Action;

// Function prototypes
void initialize();
void push(int state);
int pop();
void reduce(int prodIndex);
int findIndex(char symbol, bool isTerminal);
void parseInput();

// Global variables
int numProductions;
Production productions[10];
int numStates;
Action actionTable[10][10];
int gotoTable[10][10];
char stack[MAX_STACK_SIZE];
char input[MAX_INPUT_SIZE];
int top;
int inputIndex;

int main() {
    initialize();
    parseInput();
    return 0;
```



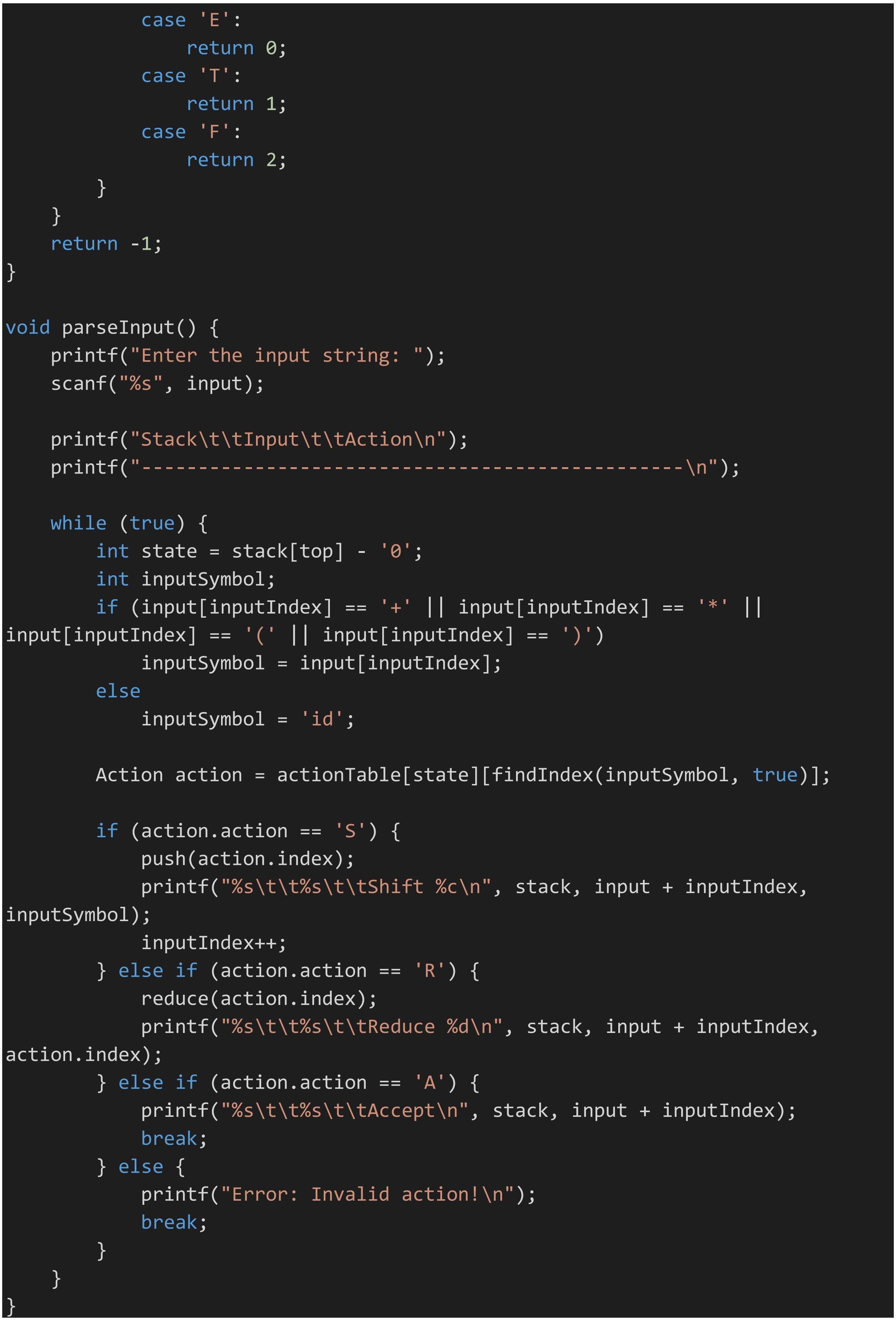
```
}

void initialize() {
    // Initialize the production rules
    numProductions = 6;
    productions[0].lhs = 'E';
    strcpy(productions[0].rhs, "E+T");
    productions[1].lhs = 'E';
    strcpy(productions[1].rhs, "T");
    productions[2].lhs = 'T';
    strcpy(productions[2].rhs, "T*F");
    productions[3].lhs = 'T';
    strcpy(productions[3].rhs, "F");
    productions[4].lhs = 'F';
    strcpy(productions[4].rhs, "(E)");
    productions[5].lhs = 'F';
    strcpy(productions[5].rhs, "id");

    // Initialize the action and goto tables
    numStates = 6;

    // State 0
    actionTable[0][0].action = ' ';
    actionTable[0][1].action = 'S';
    actionTable[0][1].index = 3;
    actionTable[0][2].action = 'S';
    actionTable[0][2].index = 4;
    actionTable[0][4].action = 'G';
    actionTable[0][4].index = 1;

    // State 1
    actionTable[1][0].action = 'R';
    actionTable[1][0].index = 1;
    actionTable[1][1].action = 'R';
    actionTable[1][1].index = 1;
    actionTable[1][2].action = 'S';
    actionTable[1][2].index = 5;
    actionTable[1][3].action = ' ';
    actionTable[1][4].action = ' ';
    actionTable[1][5].action = 'G';
    actionTable[1][5].index = 2;

    // State 2
    actionTable[2][0].action = 'R';
    actionTable[2][0].index = 3;
    actionTable[2][1].action = 'R';
    actionTable[2][1].index = 3;
    actionTable[2][2].action = 'R';
```



```
    actionTable[2][2].index = 3;
    actionTable[2][3].action = 'S';
    actionTable[2][3].index = 6;
    actionTable[2][4].action = ' ';
    actionTable[2][5].action = ' ';
    actionTable[2][6].action = ' ';

    // State 3
    actionTable[3][0].action = 'S';
    actionTable[3][0].index = 7;
    actionTable[3][1].action = 'S';
    actionTable[3][1].index = 3;
    actionTable[3][2].action = 'S';
    actionTable[3][2].index = 4;
    actionTable[3][4].action = 'G';
    actionTable[3][4].index = 8;

    // State 4
    actionTable[4][0].action = 'R';
    actionTable[4][0].index = 4;
    actionTable[4][1].action = 'R';
    actionTable[4][1].index = 4;
    actionTable[4][2].action = 'R';
    actionTable[4][2].index = 4;
    actionTable[4][3].action = 'R';
    actionTable[4][3].index = 4;
    actionTable[4][4].action = 'R';
    actionTable[4][4].index = 4;
    actionTable[4][5].action = 'R';
    actionTable[4][5].index = 4;
    actionTable[4][6].action = 'R';
    actionTable[4][6].index = 4;
    actionTable[4][7].action = 'R';
    actionTable[4][7].index = 4;
    actionTable[4][8].action = 'R';
    actionTable[4][8].index = 4;
    actionTable[4][9].action = ' ';

    // State 5
    actionTable[5][0].action = 'S';
    actionTable[5][0].index = 7;
    actionTable[5][1].action = 'S';
    actionTable[5][1].index = 3;
    actionTable[5][2].action = 'S';
    actionTable[5][2].index = 4;
    actionTable[5][4].action = 'G';
    actionTable[5][4].index = 10;
```



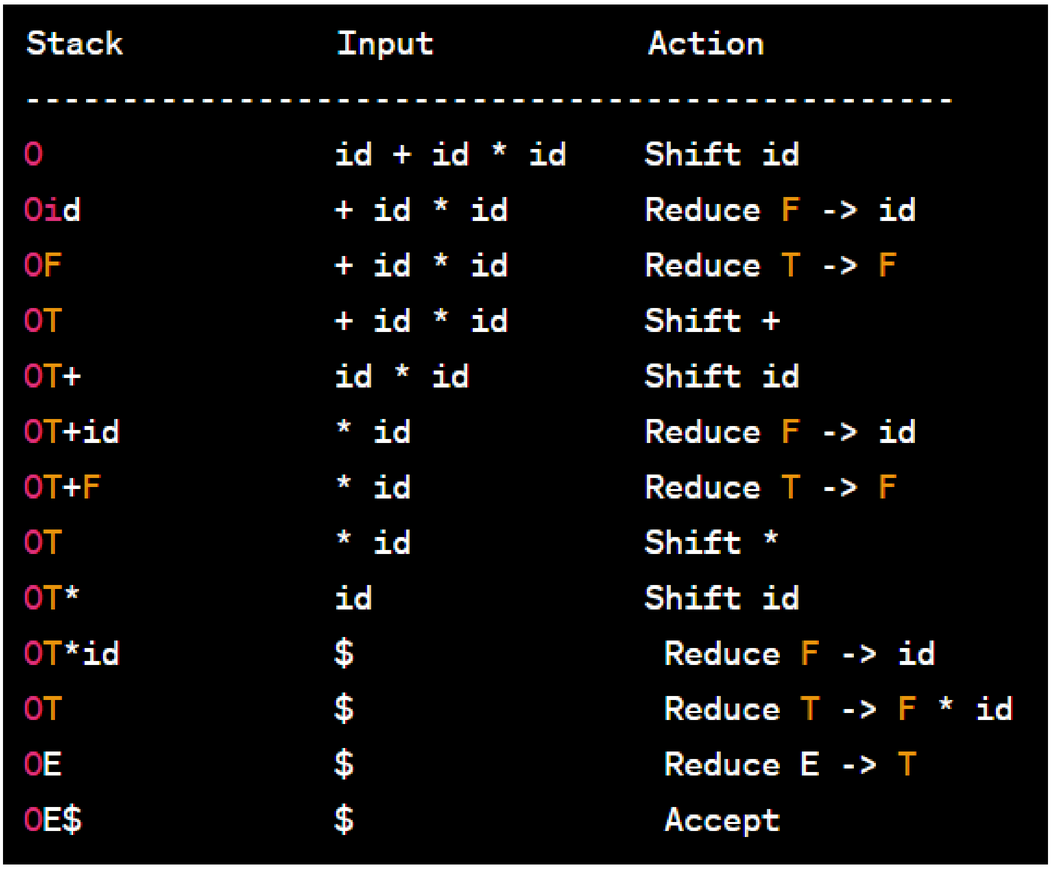
```
    // Initialize the goto table
    gotoTable[0][0] = 1;
    gotoTable[0][2] = 2;
    gotoTable[3][3] = 6;
    gotoTable[3][4] = 9;
    gotoTable[5][5] = 11;

    // Initialize stack and top
    top = -1;
    stack[++top] = '0';
}

void push(int state) {
    stack[++top] = state + '0';
}

int pop() {
    return stack[top--] - '0';
}

void reduce(int prodIndex) {
    int rhsLength = strlen(productions[prodIndex].rhs);
    while (rhsLength > 0) {
        top--;
        rhsLength--;
    }
    char lhs = productions[prodIndex].lhs;
    int state = stack[top] - '0';
    push(gotoTable[state][findIndex(lhs, false)]);
    printf("%c -> %s\n", lhs, productions[prodIndex].rhs);
}

int findIndex(char symbol, bool isTerminal) {
    if (isTerminal) {
        switch (symbol) {
            case '+':
                return 0;
            case '*':
                return 1;
            case '(':
                return 2;
            case ')':
                return 3;
            case 'id':
                return 4;
        }
    } else {
        switch (symbol) {
```



```
            case 'E':
                return 0;
            case 'T':
                return 1;
            case 'F':
                return 2;
        }
    }
    return -1;
}

void parseInput() {
    printf("Enter the input string: ");
    scanf("%s", input);

    printf("Stack\t\tInput\t\tAction\n");
    printf("------------------------------------------------\n");

    while (true) {
        int state = stack[top] - '0';
        int inputSymbol;
        if (input[inputIndex] == '+' || input[inputIndex] == '*' ||
input[inputIndex] == '(' || input[inputIndex] == ')')
            inputSymbol = input[inputIndex];
        else
            inputSymbol = 'id';

        Action action = actionTable[state][findIndex(inputSymbol, true)];

        if (action.action == 'S') {
            push(action.index);
            printf("%s\t\t%s\t\tShift %c\n", stack, input + inputIndex,
inputSymbol);
            inputIndex++;
        } else if (action.action == 'R') {
            reduce(action.index);
            printf("%s\t\t%s\t\tReduce %d\n", stack, input + inputIndex,
action.index);
        } else if (action.action == 'A') {
            printf("%s\t\t%s\t\tAccept\n", stack, input + inputIndex);
            break;
        } else {
            printf("Error: Invalid action!\n");
            break;
        }
    }
}
```

Output:

Input string: id + id * id

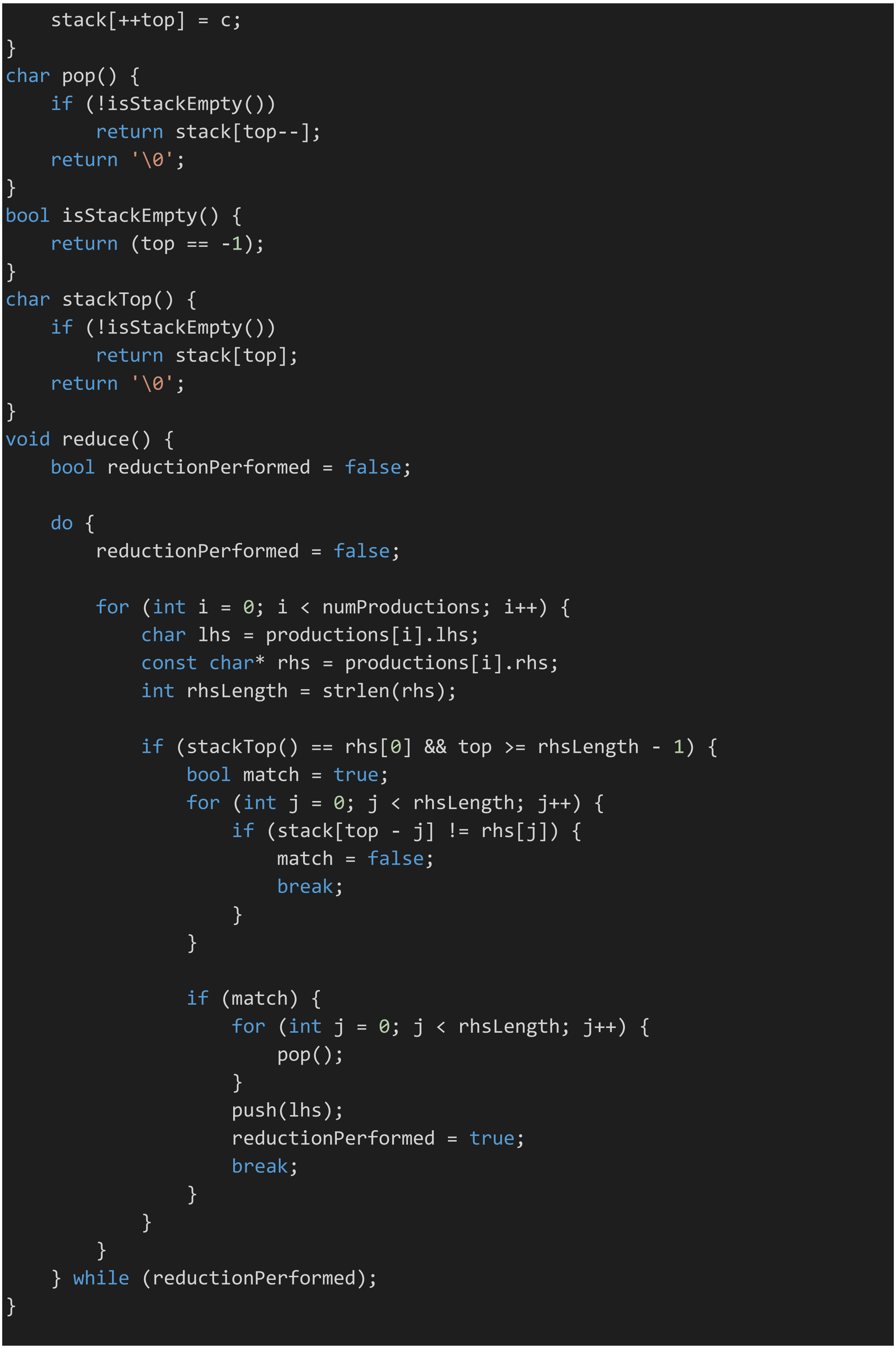


```
Stack           Input           Action
------------------------------------------------
0               id + id * id    Shift id
0id             + id * id       Reduce F -> id
0F              + id * id       Reduce T -> F
0T              + id * id       Shift +
0T+             id * id         Shift id
0T+id           * id            Reduce F -> id
0T+F            * id            Reduce T -> F
0T              * id            Shift *
0T*             id              Shift id
0T*id           $                Reduce F -> id
0T              $                Reduce T -> F * id
0E              $                Reduce E -> T
0E$             $                Accept
```

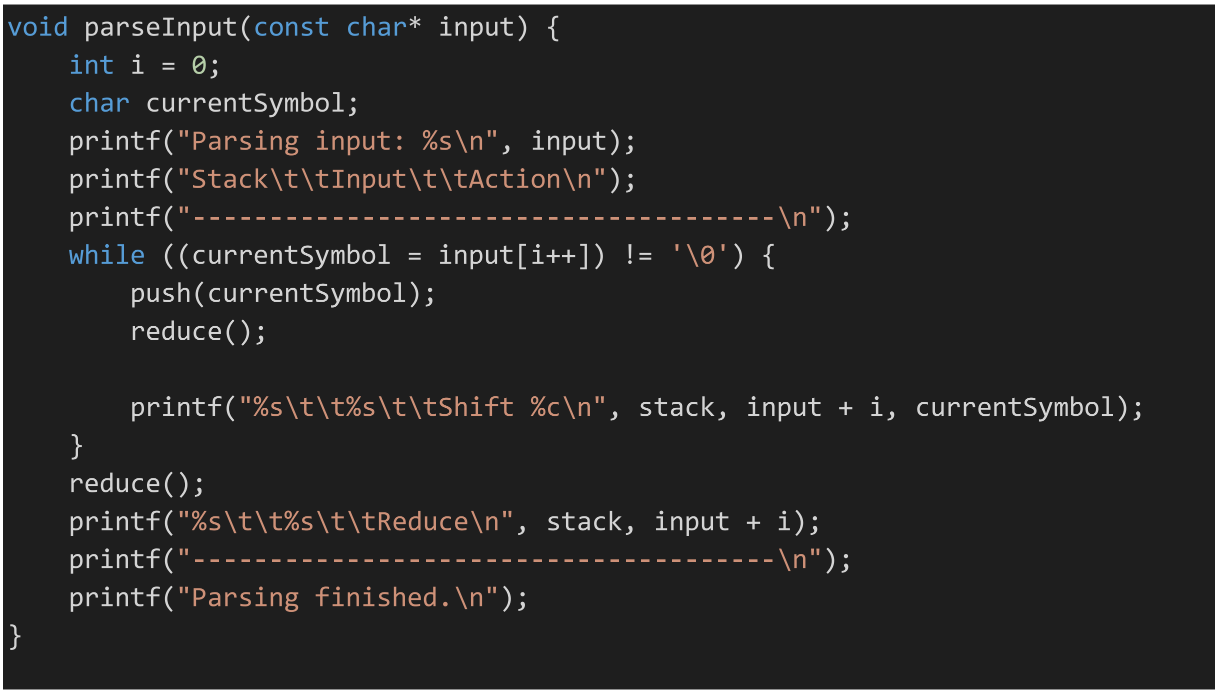
2. A C program for operator precedence parsing

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
// Structure to represent a production rule
typedef struct {
    char lhs;           // Left-hand side of the production rule
    const char* rhs;    // Right-hand side of the production rule
} Production;
// Array of production rules
Production productions[] = {
    {'E', "E+T"},
    {'E', "E-T"},
    {'E', "T"},
    {'T', "T*F"},
    {'T', "T/F"},
    {'T', "F"},
    {'F', "(E)"},
    {'F', "id"}
};
int numProductions = sizeof(productions) / sizeof(productions[0]);
// Stack implementation
#define MAX_STACK_SIZE 100
char stack[MAX_STACK_SIZE];
int top = -1;
// Function prototypes
void push(char c);
char pop();
bool isStackEmpty();
char stackTop();
void reduce();
void parseInput(const char* input);

int main() {
    // Read input from the user
    char input[100];
    printf("Enter an arithmetic expression: ");
    scanf("%s", input);
    // Parse the input expression
    parseInput(input);
    return 0;
}

void push(char c) {
```



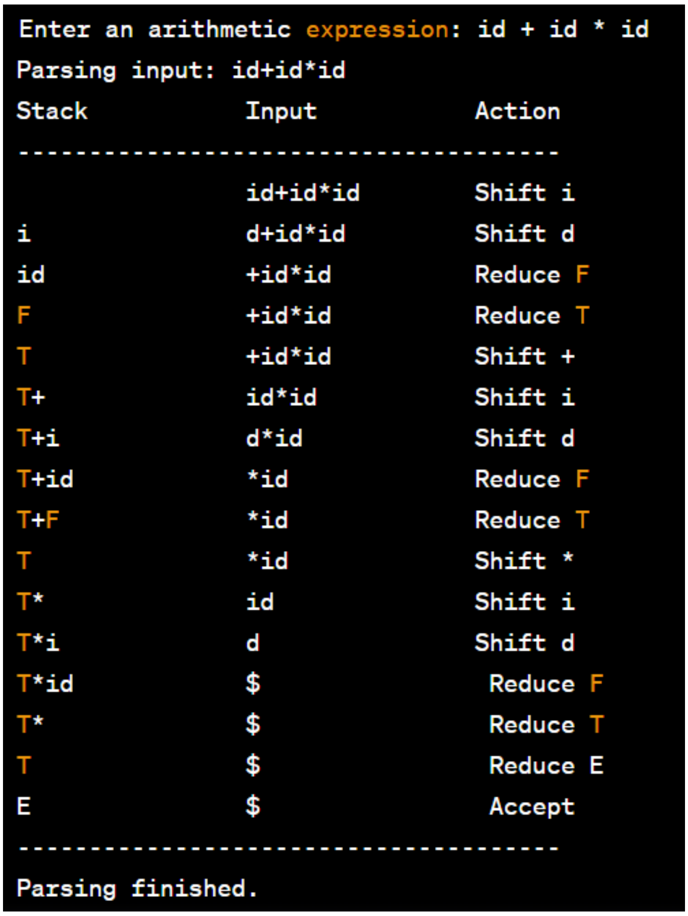
```
    stack[++top] = c;
}
char pop() {
    if (!isStackEmpty())
        return stack[top--];
    return '\0';
}
bool isStackEmpty() {
    return (top == -1);
}
char stackTop() {
    if (!isStackEmpty())
        return stack[top];
    return '\0';
}
void reduce() {
    bool reductionPerformed = false;

    do {
        reductionPerformed = false;

        for (int i = 0; i < numProductions; i++) {
            char lhs = productions[i].lhs;
            const char* rhs = productions[i].rhs;
            int rhsLength = strlen(rhs);

            if (stackTop() == rhs[0] && top >= rhsLength - 1) {
                bool match = true;
                for (int j = 0; j < rhsLength; j++) {
                    if (stack[top - j] != rhs[j]) {
                        match = false;
                        break;
                    }
                }

                if (match) {
                    for (int j = 0; j < rhsLength; j++) {
                        pop();
                    }
                    push(lhs);
                    reductionPerformed = true;
                    break;
                }
            }
        }
    } while (reductionPerformed);
}
```



```c
void parseInput(const char* input) {
    int i = 0;
    char currentSymbol;
    printf("Parsing input: %s\n", input);
    printf("Stack\t\tInput\t\tAction\n");
    printf("--------------------------------------\n");
    while ((currentSymbol = input[i++]) != '\0') {
        push(currentSymbol);
        reduce();

        printf("%s\t\t%s\t\tShift %c\n", stack, input + i, currentSymbol);
    }
    reduce();
    printf("%s\t\t%s\t\tReduce\n", stack, input + i);
    printf("--------------------------------------\n");
    printf("Parsing finished.\n");
}
```

Output:

Input string: id + id * id



```
Enter an arithmetic expression: id + id * id
Parsing input: id+id*id
Stack           Input           Action
--------------------------------------
                id+id*id        Shift i
i               d+id*id         Shift d
id              +id*id          Reduce F
F               +id*id          Reduce T
T               +id*id          Shift +
T+              id*id           Shift i
T+i             d*id            Shift d
T+id            *id             Reduce F
T+F             *id             Reduce T
T               *id             Shift *
T*              id              Shift i
T*i             d               Shift d
T*id            $                Reduce F
T*              $                Reduce T
T               $                Reduce E
E               $                Accept
--------------------------------------
Parsing finished.
```