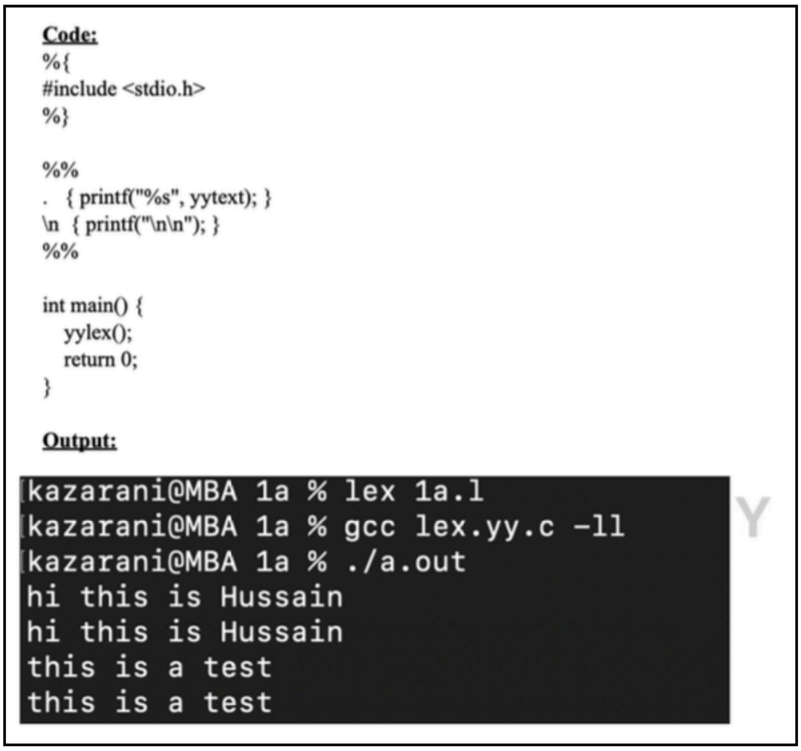
CD Lab

1.1)



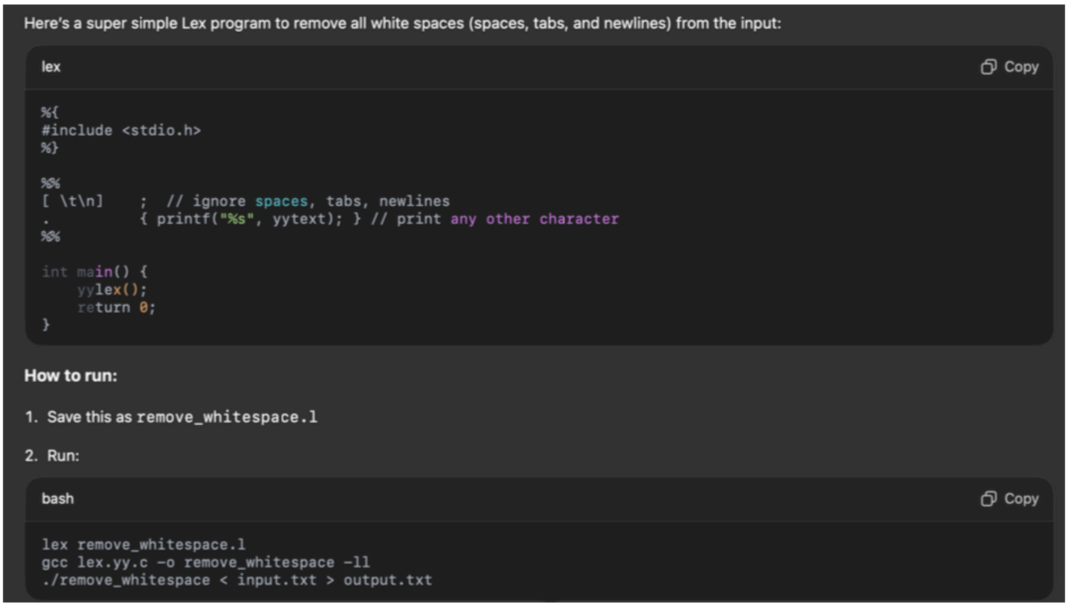
Code:

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

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

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

Output:

```
kazarani@MBA 1a % lex 1a.l
kazarani@MBA 1a % gcc lex.yy.c -ll
kazarani@MBA 1a % ./a.out
hi this is Hussain
hi this is Hussain
this is a test
this is a test
```

1.2)



Here's a super simple Lex program to remove all white spaces (spaces, tabs, and newlines) from the input:

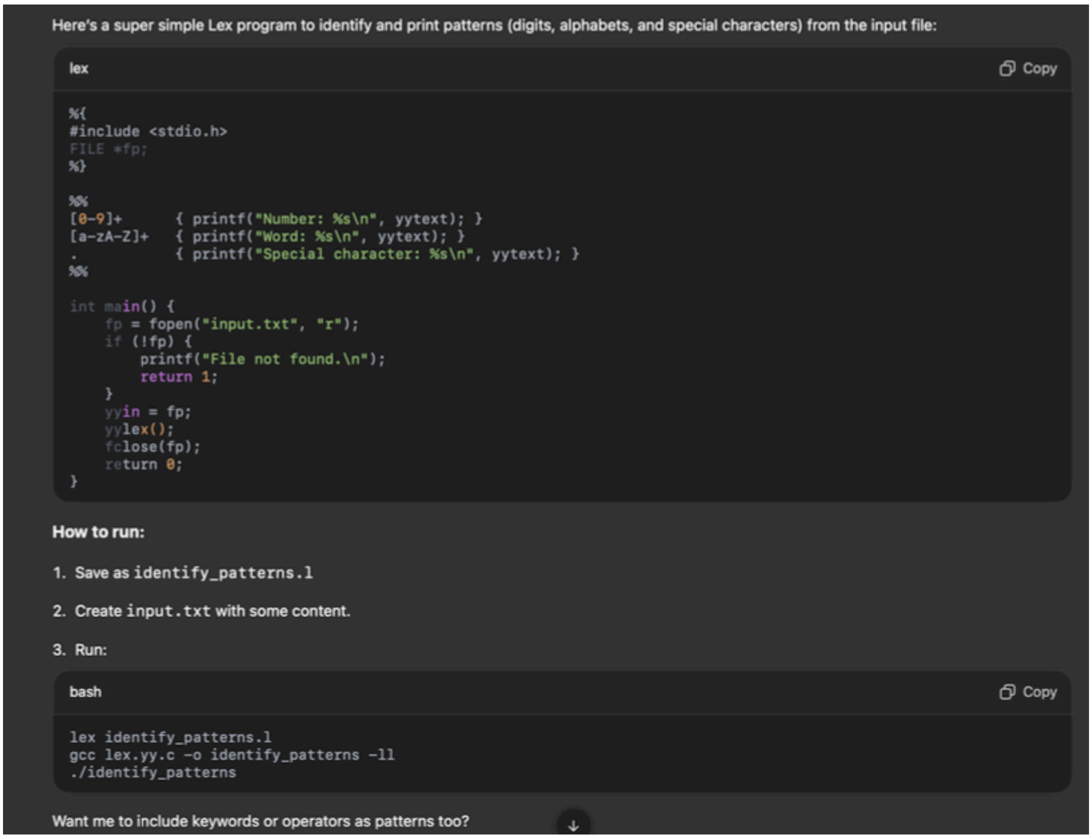
```
%{
#include <stdio.h>
%}

%%
[ \t\n]    ;  // ignore spaces, tabs, newlines
.          { printf("%s", yytext); } // print any other character
%%

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

How to run:

1. Save this as `remove_whitespace.l`
2. Run:

```
lex remove_whitespace.l
gcc lex.yy.c -o remove_whitespace -ll
./remove_whitespace < input.txt > output.txt
```

2.1)



Here's a super simple Lex program to identify and print patterns (digits, alphabets, and special characters) from the input file:

```
%{
#include <stdio.h>
FILE *fp;
%}

%%
[0-9]+      { printf("Number: %s\n", yytext); }
[a-zA-Z]+   { printf("Word: %s\n", yytext); }
.           { printf("Special character: %s\n", yytext); }
%%

int main() {
    fp = fopen("input.txt", "r");
    if (!fp) {
        printf("File not found.\n");
        return 1;
    }
    yyin = fp;
    yylex();
    fclose(fp);
    return 0;
}
```

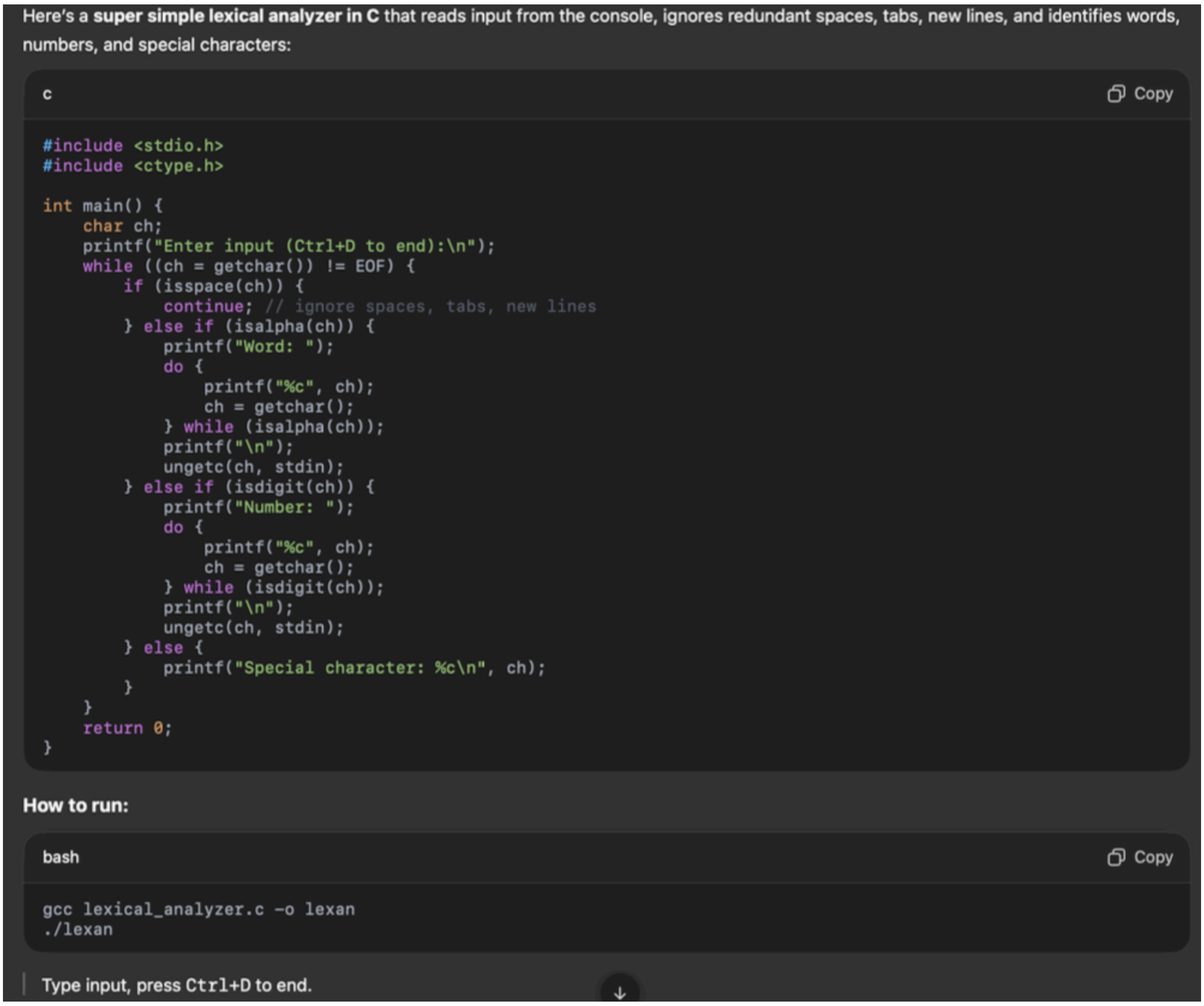
How to run:

1. Save as `identify_patterns.l`
2. Create `input.txt` with some content.
3. Run:

```
lex identify_patterns.l
gcc lex.yy.c -o identify_patterns -ll
./identify_patterns
```

Want me to include keywords or operators as patterns too?

2.2)



Here's a **super simple lexical analyzer in C** that reads input from the console, ignores redundant spaces, tabs, new lines, and identifies words, numbers, and special characters:

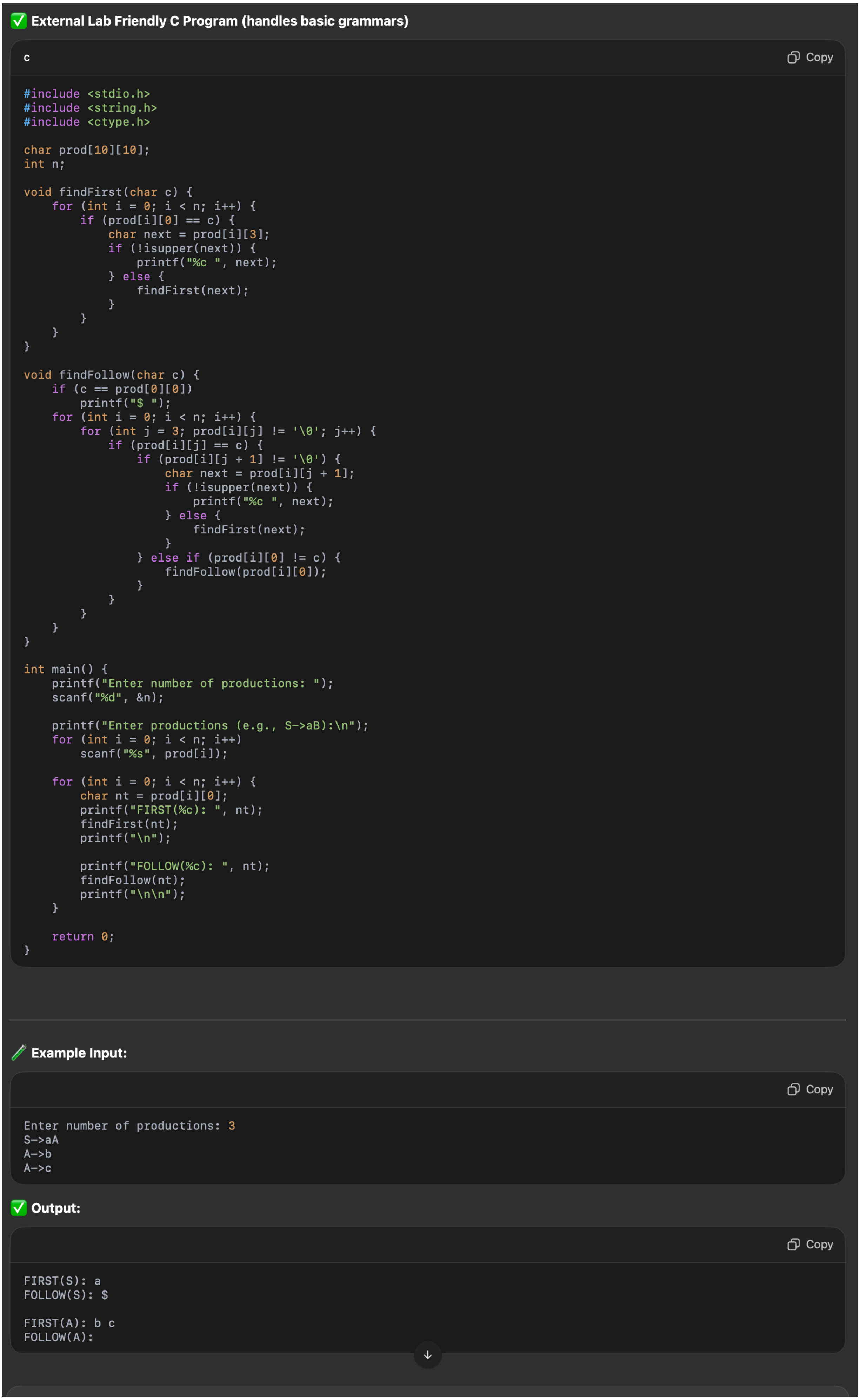
```
#include <stdio.h>
#include <ctype.h>

int main() {
    char ch;
    printf("Enter input (Ctrl+D to end):\n");
    while ((ch = getchar()) != EOF) {
        if (isspace(ch)) {
            continue; // ignore spaces, tabs, new lines
        } else if (isalpha(ch)) {
            printf("Word: ");
            do {
                printf("%c", ch);
                ch = getchar();
            } while (isalpha(ch));
            printf("\n");
            ungetc(ch, stdin);
        } else if (isdigit(ch)) {
            printf("Number: ");
            do {
                printf("%c", ch);
                ch = getchar();
            } while (isdigit(ch));
            printf("\n");
            ungetc(ch, stdin);
        } else {
            printf("Special character: %c\n", ch);
        }
    }
    return 0;
}
```

How to run:

```
gcc lexical_analyzer.c -o lexan
./lexan
```

Type input, press `Ctrl+D` to end.

3.1)



✅ **External Lab Friendly C Program (handles basic grammars)**

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

char prod[10][10];
int n;

void findFirst(char c) {
    for (int i = 0; i < n; i++) {
        if (prod[i][0] == c) {
            char next = prod[i][3];
            if (!isupper(next)) {
                printf("%c ", next);
            } else {
                findFirst(next);
            }
        }
    }
}

void findFollow(char c) {
    if (c == prod[0][0])
        printf("$ ");
    for (int i = 0; i < n; i++) {
        for (int j = 3; prod[i][j] != '\0'; j++) {
            if (prod[i][j] == c) {
                if (prod[i][j + 1] != '\0') {
                    char next = prod[i][j + 1];
                    if (!isupper(next)) {
                        printf("%c ", next);
                    } else {
                        findFirst(next);
                    }
                } else if (prod[i][0] != c) {
                    findFollow(prod[i][0]);
                }
            }
        }
    }
}

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

    printf("Enter productions (e.g., S->aB):\n");
    for (int i = 0; i < n; i++)
        scanf("%s", prod[i]);

    for (int i = 0; i < n; i++) {
        char nt = prod[i][0];
        printf("FIRST(%c): ", nt);
        findFirst(nt);
        printf("\n");

        printf("FOLLOW(%c): ", nt);
        findFollow(nt);
        printf("\n\n");
    }

    return 0;
}
```

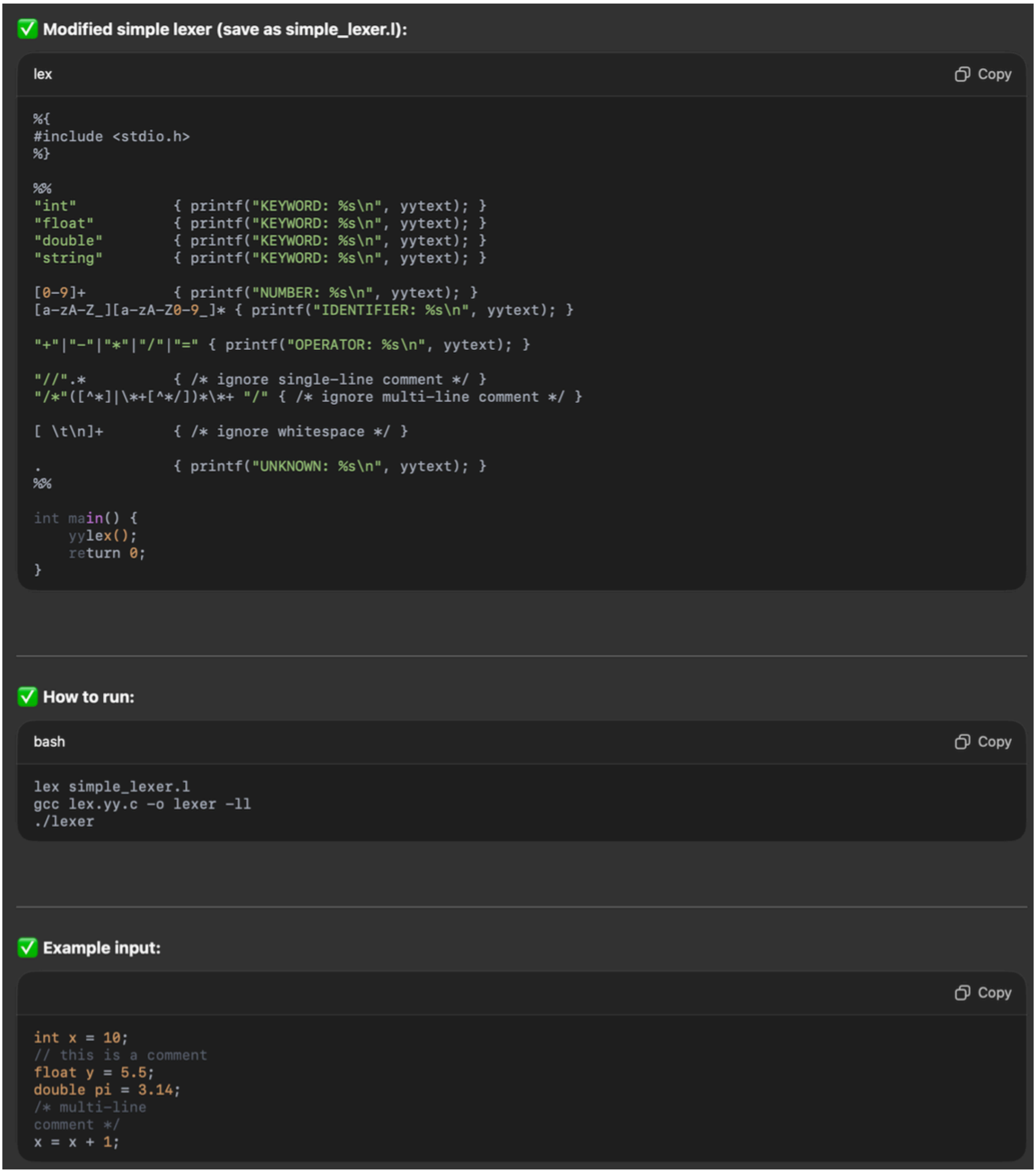
🧪 **Example Input:**

```
Enter number of productions: 3
S->aA
A->b
A->c
```

✅ **Output:**

```
FIRST(S): a
FOLLOW(S): $

FIRST(A): b c
FOLLOW(A):
```

3.2)



✅ **Modified simple lexer (save as simple_lexer.l):**

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

%%
"int"           { printf("KEYWORD: %s\n", yytext); }
"float"         { printf("KEYWORD: %s\n", yytext); }
"double"        { printf("KEYWORD: %s\n", yytext); }
"string"        { printf("KEYWORD: %s\n", yytext); }

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

"+"|"-"|"*"|"/"|"=" { printf("OPERATOR: %s\n", yytext); }

"//".*          { /* ignore single-line comment */ }
"/*"([^*]|\*+[^*/])*\*+ "/" { /* ignore multi-line comment */ }

[ \t\n]+        { /* ignore whitespace */ }

.               { printf("UNKNOWN: %s\n", yytext); }
%%

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

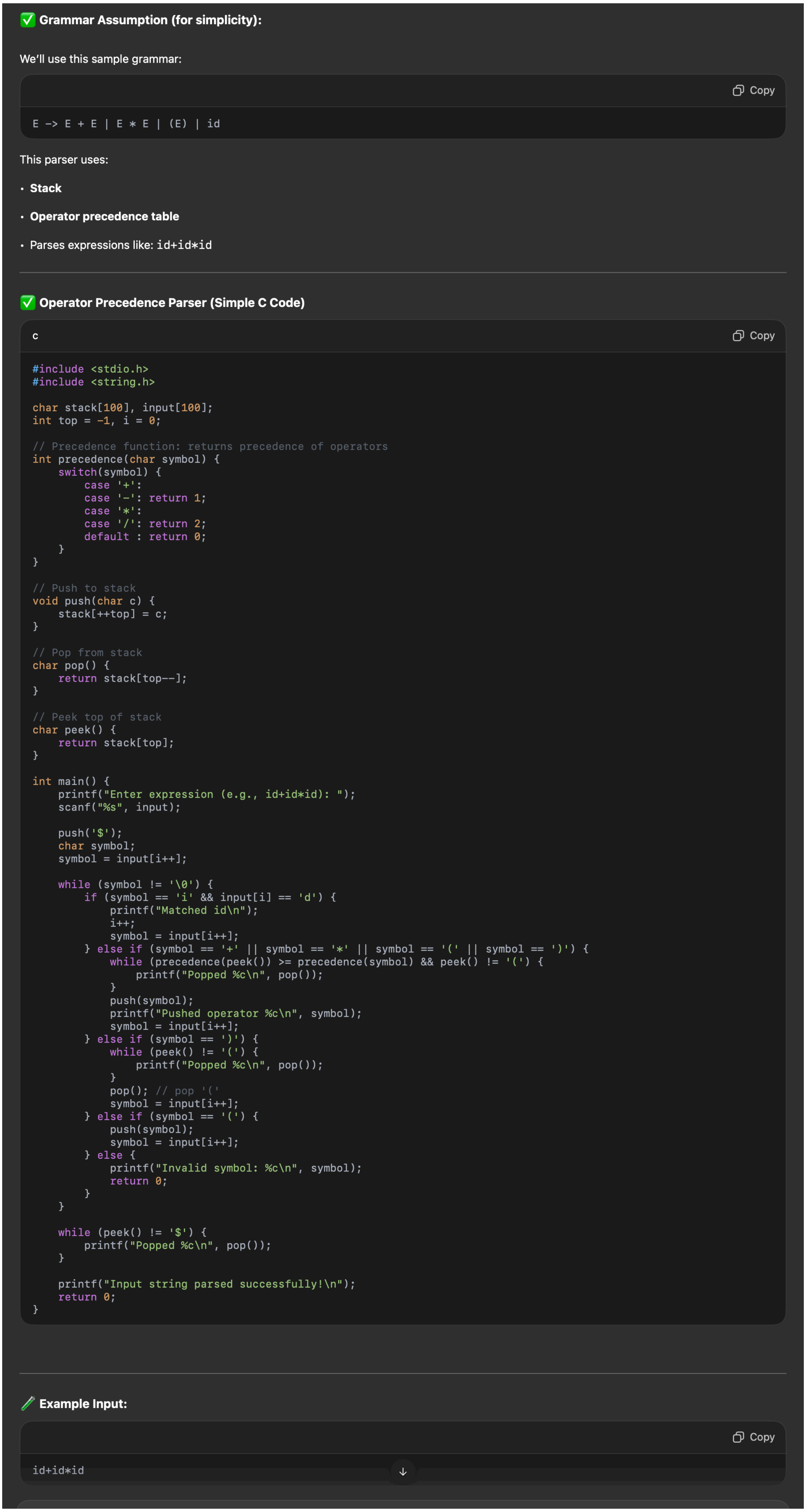
✅ **How to run:**

```bash
lex simple_lexer.l
gcc lex.yy.c -o lexer -ll
./lexer
```

✅ **Example input:**

```
int x = 10;
// this is a comment
float y = 5.5;
double pi = 3.14;
/* multi-line
comment */
x = x + 1;
```

4.1)



✅ **Grammar Assumption (for simplicity):**

We'll use this sample grammar:

```
E -> E + E | E * E | (E) | id
```

This parser uses:

- **Stack**
- **Operator precedence table**
- Parses expressions like: `id+id*id`

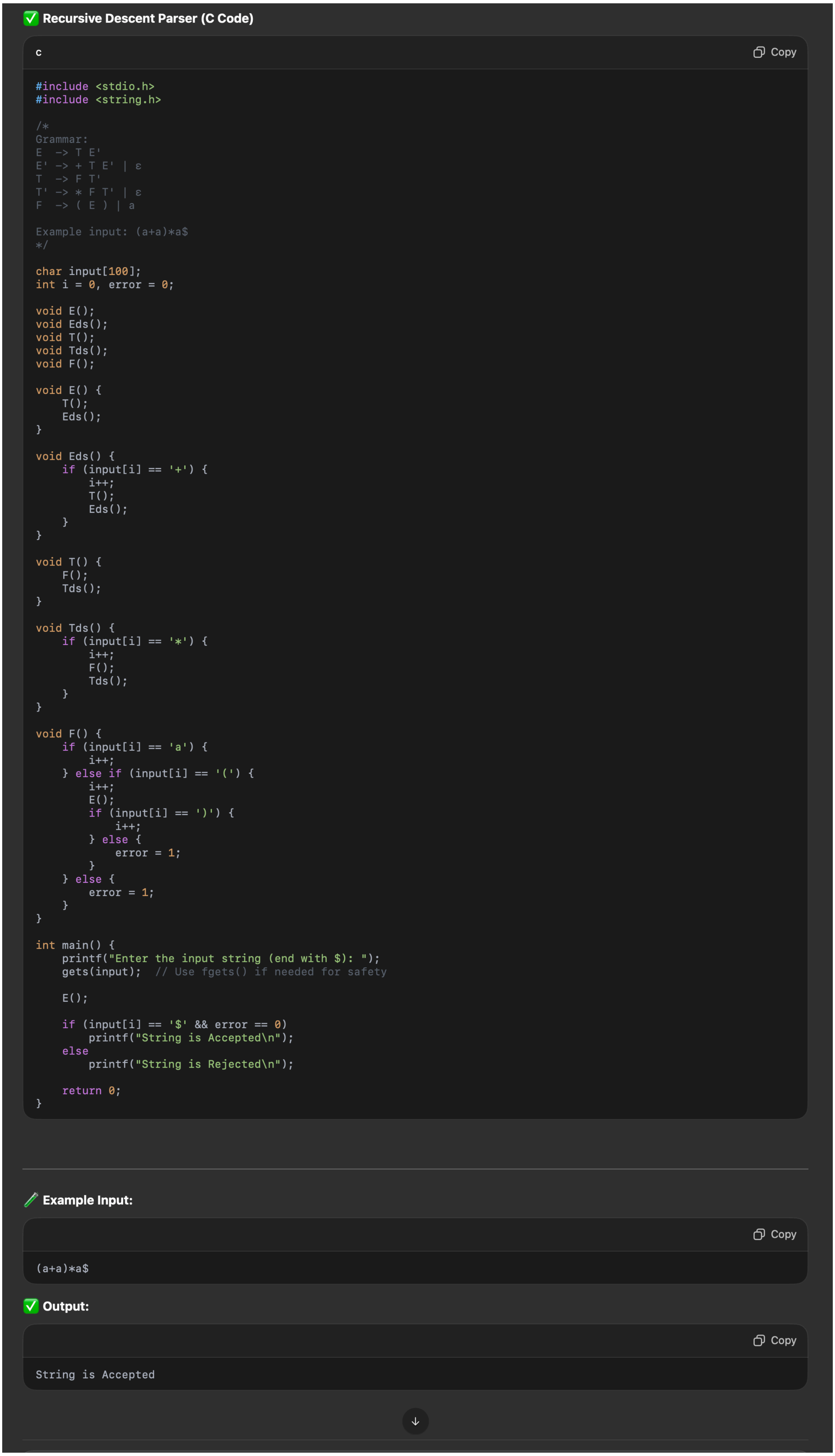
✅ **Operator Precedence Parser (Simple C Code)**

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

char stack[100], input[100];
int top = -1, i = 0;

// Precedence function: returns precedence of operators
int precedence(char symbol) {
    switch(symbol) {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 2;
        default : return 0;
    }
}

// Push to stack
void push(char c) {
    stack[++top] = c;
}

// Pop from stack
char pop() {
    return stack[top--];
}

// Peek top of stack
char peek() {
    return stack[top];
}

int main() {
    printf("Enter expression (e.g., id+id*id): ");
    scanf("%s", input);

    push('$');
    char symbol;
    symbol = input[i++];

    while (symbol != '\0') {
        if (symbol == 'i' && input[i] == 'd') {
            printf("Matched id\n");
            i++;
            symbol = input[i++];
        } else if (symbol == '+' || symbol == '*' || symbol == '(' || symbol == ')') {
            while (precedence(peek()) >= precedence(symbol) && peek() != '(') {
                printf("Popped %c\n", pop());
            }
            push(symbol);
            printf("Pushed operator %c\n", symbol);
            symbol = input[i++];
        } else if (symbol == ')') {
            while (peek() != '(') {
                printf("Popped %c\n", pop());
            }
            pop(); // pop '('
            symbol = input[i++];
        } else if (symbol == '(') {
            push(symbol);
            symbol = input[i++];
        } else {
            printf("Invalid symbol: %c\n", symbol);
            return 0;
        }
    }

    while (peek() != '$') {
        printf("Popped %c\n", pop());
    }

    printf("Input string parsed successfully!\n");
    return 0;
}
```

🧪 **Example Input:**

```
id+id*id
```

4.2)



✅ **Recursive Descent Parser (C Code)**

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

/*
Grammar:
E  -> T E'
E' -> + T E' | ε
T  -> F T'
T' -> * F T' | ε
F  -> ( E ) | a

Example input: (a+a)*a$
*/

char input[100];
int i = 0, error = 0;

void E();
void Eds();
void T();
void Tds();
void F();

void E() {
    T();
    Eds();
}

void Eds() {
    if (input[i] == '+') {
        i++;
        T();
        Eds();
    }
}

void T() {
    F();
    Tds();
}

void Tds() {
    if (input[i] == '*') {
        i++;
        F();
        Tds();
    }
}

void F() {
    if (input[i] == 'a') {
        i++;
    } else if (input[i] == '(') {
        i++;
        E();
        if (input[i] == ')') {
            i++;
        } else {
            error = 1;
        }
    } else {
        error = 1;
    }
}

int main() {
    printf("Enter the input string (end with $): ");
    gets(input);  // Use fgets() if needed for safety

    E();

    if (input[i] == '$' && error == 0)
        printf("String is Accepted\n");
    else
        printf("String is Rejected\n");

    return 0;
}
```

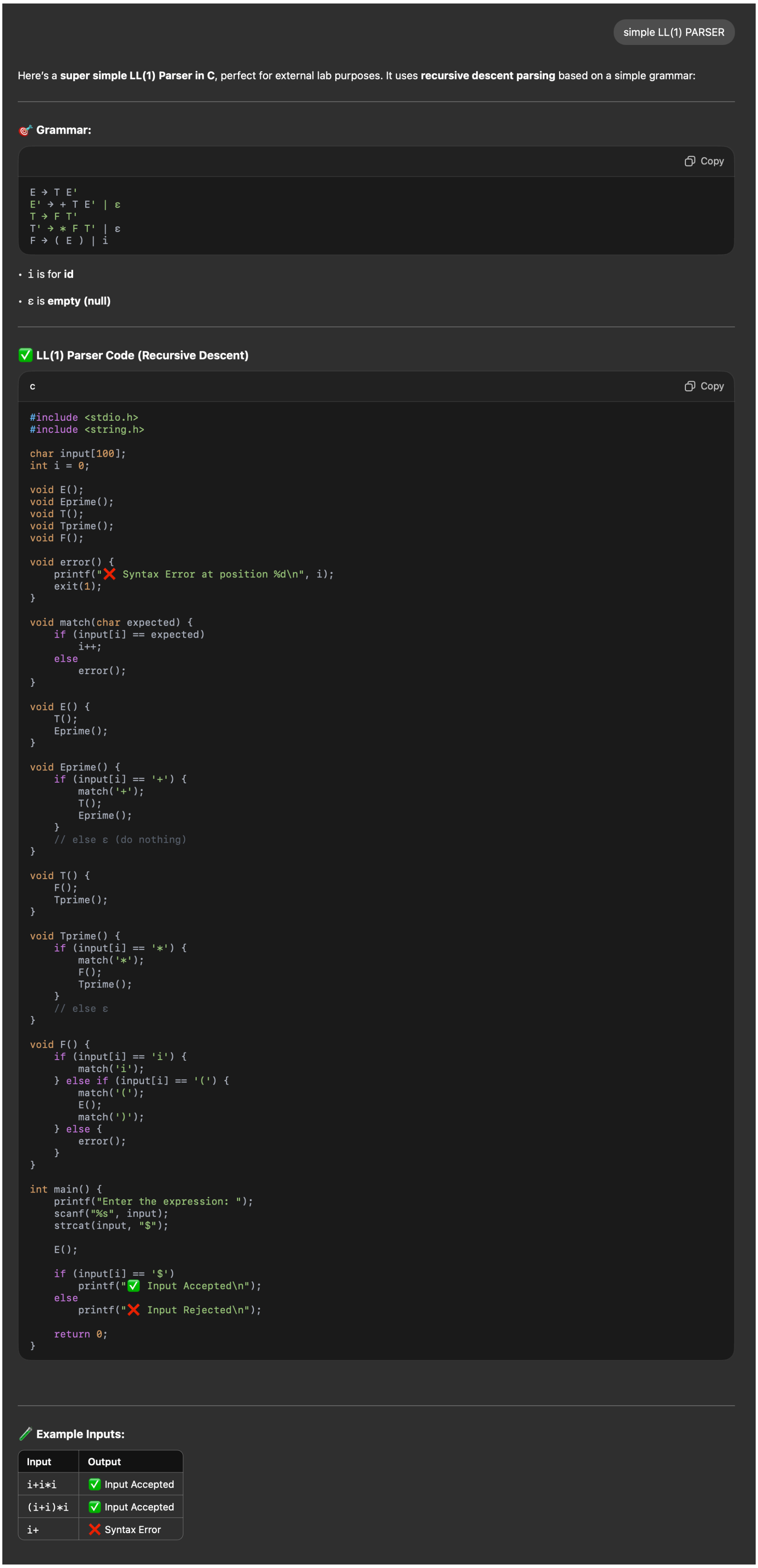
🖊️ **Example Input:**

```
(a+a)*a$
```

✅ **Output:**

```
String is Accepted
```

5.1)



Here's a **super simple LL(1) Parser in C**, perfect for external lab purposes. It uses **recursive descent parsing** based on a simple grammar:

🎯 **Grammar:**

```
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → ( E ) | i
```

- i is for **id**
- ε is **empty (null)**

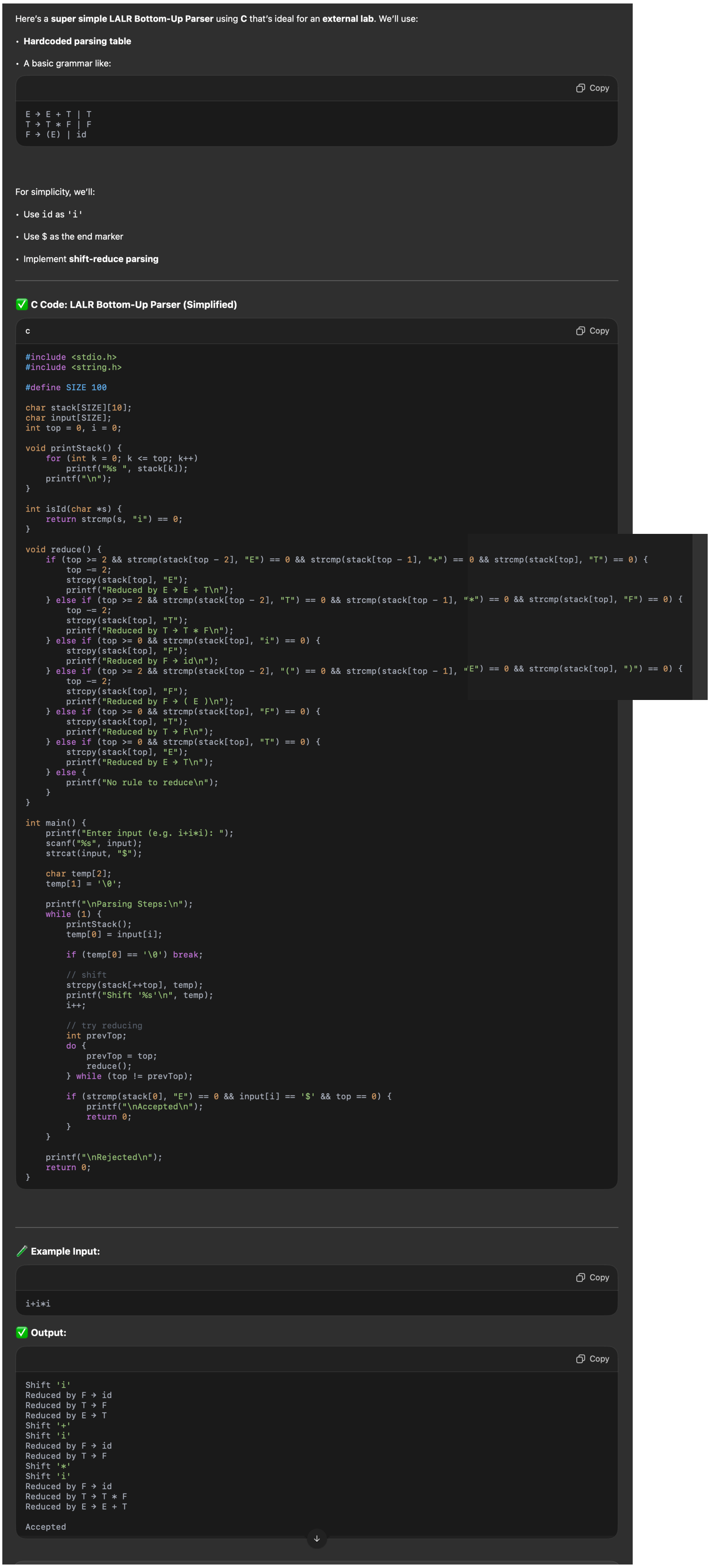
✅ **LL(1) Parser Code (Recursive Descent)**

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

char input[100];
int i = 0;

void E();
void Eprime();
void T();
void Tprime();
void F();

void error() {
    printf("❌ Syntax Error at position %d\n", i);
    exit(1);
}

void match(char expected) {
    if (input[i] == expected)
        i++;
    else
        error();
}

void E() {
    T();
    Eprime();
}

void Eprime() {
    if (input[i] == '+') {
        match('+');
        T();
        Eprime();
    }
    // else ε (do nothing)
}

void T() {
    F();
    Tprime();
}

void Tprime() {
    if (input[i] == '*') {
        match('*');
        F();
        Tprime();
    }
    // else ε
}

void F() {
    if (input[i] == 'i') {
        match('i');
    } else if (input[i] == '(') {
        match('(');
        E();
        match(')');
    } else {
        error();
    }
}

int main() {
    printf("Enter the expression: ");
    scanf("%s", input);
    strcat(input, "$");

    E();

    if (input[i] == '$')
        printf("✅ Input Accepted\n");
    else
        printf("❌ Input Rejected\n");

    return 0;
}
```

🧪 **Example Inputs:**

Input	Output
i+i*i	✅ Input Accepted
(i+i)*i	✅ Input Accepted
i+	❌ Syntax Error

5.2)



Here's a super simple **LALR Bottom-Up Parser** using **C** that's ideal for an **external lab**. We'll use:

- **Hardcoded parsing table**
- A basic grammar like:

```
E → E + T | T
T → T * F | F
F → (E) | id
```

For simplicity, we'll:

- Use id as `'i'`
- Use $ as the end marker
- Implement **shift-reduce parsing**

✅ **C Code: LALR Bottom-Up Parser (Simplified)**

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

#define SIZE 100

char stack[SIZE][10];
char input[SIZE];
int top = 0, i = 0;

void printStack() {
    for (int k = 0; k <= top; k++)
        printf("%s ", stack[k]);
    printf("\n");
}

int isId(char *s) {
    return strcmp(s, "i") == 0;
}

void reduce() {
    if (top >= 2 && strcmp(stack[top - 2], "E") == 0 && strcmp(stack[top - 1], "+") == 0 && strcmp(stack[top], "T") == 0) {
        top -= 2;
        strcpy(stack[top], "E");
        printf("Reduced by E → E + T\n");
    } else if (top >= 2 && strcmp(stack[top - 2], "T") == 0 && strcmp(stack[top - 1], "*") == 0 && strcmp(stack[top], "F") == 0) {
        top -= 2;
        strcpy(stack[top], "T");
        printf("Reduced by T → T * F\n");
    } else if (top >= 0 && strcmp(stack[top], "i") == 0) {
        strcpy(stack[top], "F");
        printf("Reduced by F → id\n");
    } else if (top >= 2 && strcmp(stack[top - 2], "(") == 0 && strcmp(stack[top - 1], "E") == 0 && strcmp(stack[top], ")") == 0) {
        top -= 2;
        strcpy(stack[top], "F");
        printf("Reduced by F → ( E )\n");
    } else if (top >= 0 && strcmp(stack[top], "F") == 0) {
        strcpy(stack[top], "T");
        printf("Reduced by T → F\n");
    } else if (top >= 0 && strcmp(stack[top], "T") == 0) {
        strcpy(stack[top], "E");
        printf("Reduced by E → T\n");
    } else {
        printf("No rule to reduce\n");
    }
}

int main() {
    printf("Enter input (e.g. i+i*i): ");
    scanf("%s", input);
    strcat(input, "$");

    char temp[2];
    temp[1] = '\0';

    printf("\nParsing Steps:\n");
    while (1) {
        printStack();
        temp[0] = input[i];

        if (temp[0] == '\0') break;

        // shift
        strcpy(stack[++top], temp);
        printf("Shift '%s'\n", temp);
        i++;

        // try reducing
        int prevTop;
        do {
            prevTop = top;
            reduce();
        } while (top != prevTop);

        if (strcmp(stack[0], "E") == 0 && input[i] == '$' && top == 0) {
            printf("\nAccepted\n");
            return 0;
        }
    }

    printf("\nRejected\n");
    return 0;
}
```

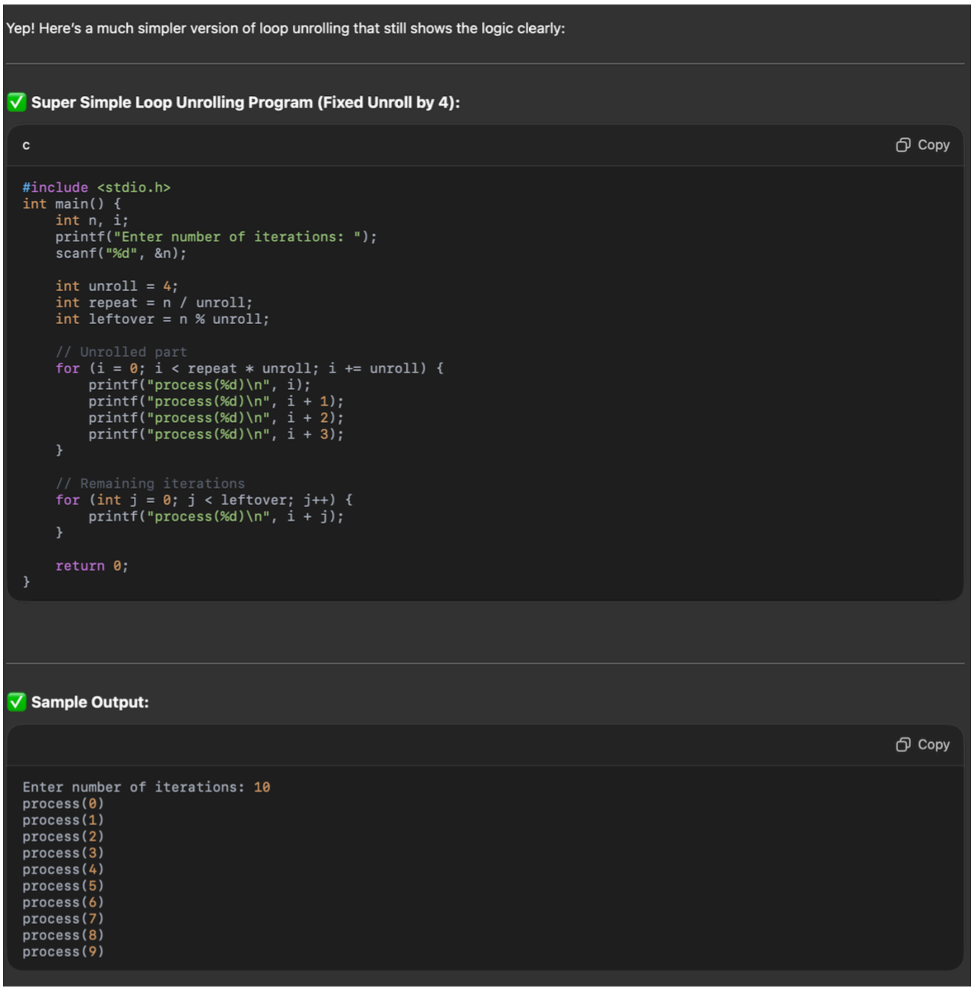
🧪 **Example Input:**

```
i+i*i
```

✅ **Output:**

```
Shift 'i'
Reduced by F → id
Reduced by T → F
Reduced by E → T
Shift '+'
Shift 'i'
Reduced by F → id
Reduced by T → F
Shift '*'
Shift 'i'
Reduced by F → id
Reduced by T → T * F
Reduced by E → E + T

Accepted
```

6.1)



Yep! Here's a much simpler version of loop unrolling that still shows the logic clearly:

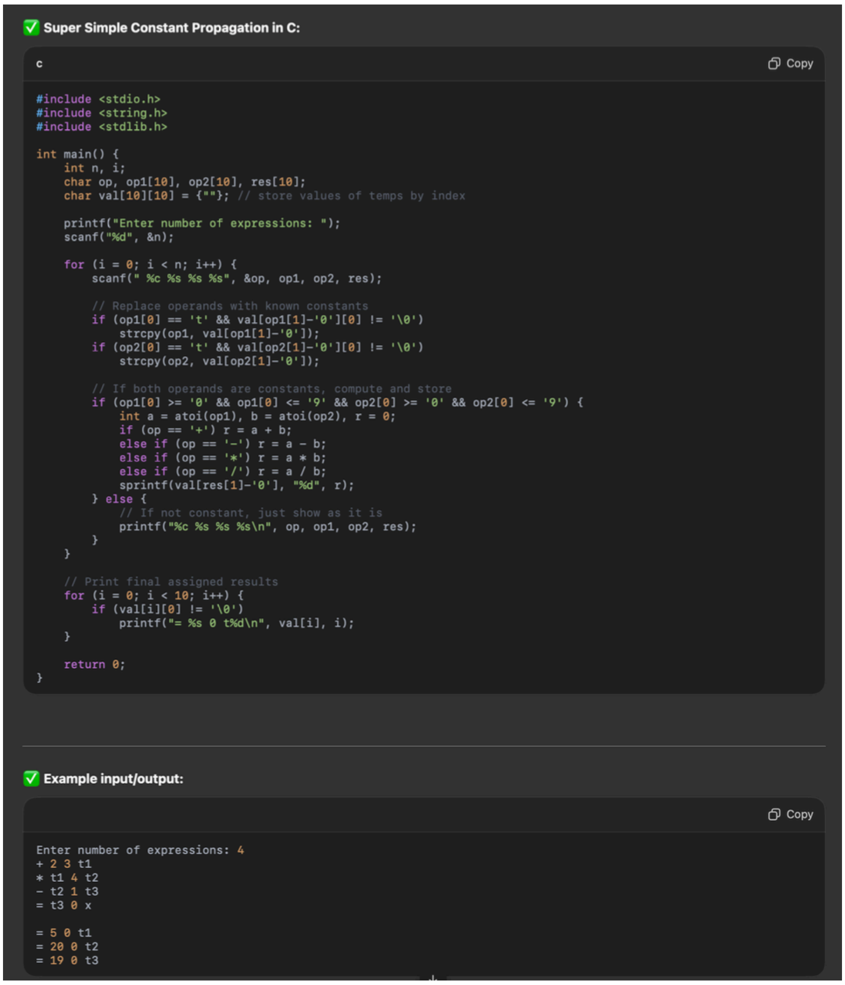
✅ **Super Simple Loop Unrolling Program (Fixed Unroll by 4):**

```c
#include <stdio.h>
int main() {
    int n, i;
    printf("Enter number of iterations: ");
    scanf("%d", &n);

    int unroll = 4;
    int repeat = n / unroll;
    int leftover = n % unroll;

    // Unrolled part
    for (i = 0; i < repeat * unroll; i += unroll) {
        printf("process(%d)\n", i);
        printf("process(%d)\n", i + 1);
        printf("process(%d)\n", i + 2);
        printf("process(%d)\n", i + 3);
    }

    // Remaining iterations
    for (int j = 0; j < leftover; j++) {
        printf("process(%d)\n", i + j);
    }

    return 0;
}
```

✅ **Sample Output:**

```
Enter number of iterations: 10
process(0)
process(1)
process(2)
process(3)
process(4)
process(5)
process(6)
process(7)
process(8)
process(9)
```

6.2)



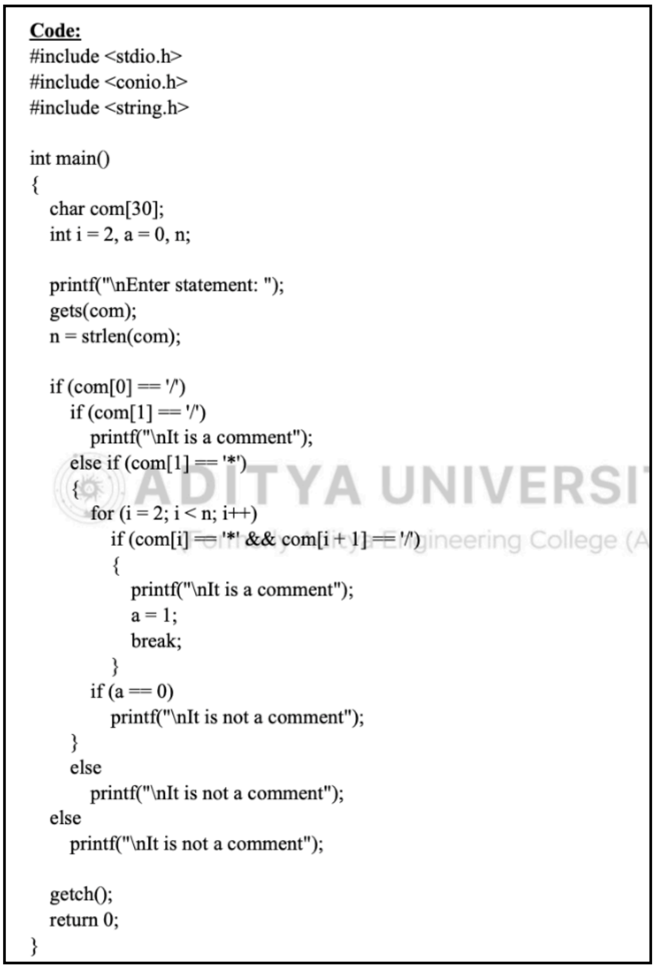
✅ **Super Simple Constant Propagation in C:**

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

int main() {
    int n, i;
    char op, op1[10], op2[10], res[10];
    char val[10][10] = {""}; // store values of temps by index

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

    for (i = 0; i < n; i++) {
        scanf(" %c %s %s %s", &op, op1, op2, res);

        // Replace operands with known constants
        if (op1[0] == 't' && val[op1[1]-'0'][0] != '\0')
            strcpy(op1, val[op1[1]-'0']);
        if (op2[0] == 't' && val[op2[1]-'0'][0] != '\0')
            strcpy(op2, val[op2[1]-'0']);

        // If both operands are constants, compute and store
        if (op1[0] >= '0' && op1[0] <= '9' && op2[0] >= '0' && op2[0] <= '9') {
            int a = atoi(op1), b = atoi(op2), r = 0;
            if (op == '+') r = a + b;
            else if (op == '-') r = a - b;
            else if (op == '*') r = a * b;
            else if (op == '/') r = a / b;
            sprintf(val[res[1]-'0'], "%d", r);
        } else {
            // If not constant, just show as it is
            printf("%c %s %s %s\n", op, op1, op2, res);
        }
    }

    // Print final assigned results
    for (i = 0; i < 10; i++) {
        if (val[i][0] != '\0')
            printf("= %s 0 t%d\n", val[i], i);
    }

    return 0;
}
```

✅ **Example input/output:**

```
Enter number of expressions: 4
+ 2 3 t1
* t1 4 t2
- t2 1 t3
= t3 0 x

= 5 0 t1
= 20 0 t2
= 19 0 t3
```

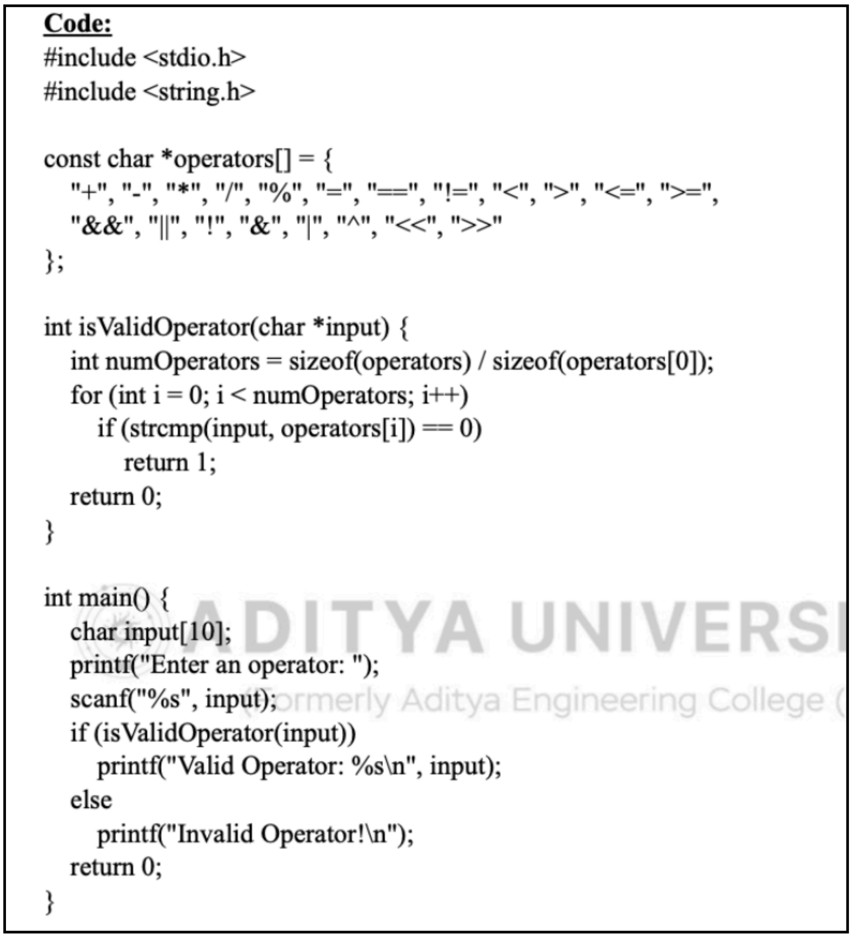
13)



Code:

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

int main()
{
  char com[30];
  int i = 2, a = 0, n;

  printf("\nEnter statement: ");
  gets(com);
  n = strlen(com);

  if (com[0] == '/')
    if (com[1] == '/')
      printf("\nIt is a comment");
    else if (com[1] == '*')
    {
      for (i = 2; i < n; i++)
        if (com[i] == '*' && com[i + 1] == '/')
        {
          printf("\nIt is a comment");
          a = 1;
          break;
        }
      if (a == 0)
        printf("\nIt is not a comment");
    }
    else
      printf("\nIt is not a comment");
  else
    printf("\nIt is not a comment");

  getch();
  return 0;
}
```

14)



Code:

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

const char *operators[] = {
  "+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">=",
  "&&", "||", "!", "&", "|", "^", "<<", ">>"
};

int isValidOperator(char *input) {
  int numOperators = sizeof(operators) / sizeof(operators[0]);
  for (int i = 0; i < numOperators; i++)
    if (strcmp(input, operators[i]) == 0)
      return 1;
  return 0;
}

int main() {
  char input[10];
  printf("Enter an operator: ");
  scanf("%s", input);
  if (isValidOperator(input))
    printf("Valid Operator: %s\n", input);
  else
    printf("Invalid Operator!\n");
  return 0;
}
```