NIRMA UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED ‘A’ GRADE

B. Tech. Semester VII
Academic Year 2021-22
Computer Science and Engineering

2CS701 - Compiler Construction
Innovative Assignment

Project Title:- Mini C Compiler

Submitted by:
18BCE155 - Cherish Patel
18BCE164 - Meet Patel

Submitted to:
Prof. Monika Shah

Introduction

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programming mistakes. When executing the code, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of the source program, and feeds its output to the next phase of the compiler. We basically have two phases of compilers, namely Analysis phase and Synthesis phase.

Analysis phase creates an intermediate representation from the given source code. It is also termed as the front end of the compiler. We have implemented first 3 stages of Analysis phase that are as follows:

1. Lexical Analysis :

- The lexical phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as, an identifier, a keyword, a punctuation character.
- The character sequence forming a token is called the lexeme for the token.

2. Syntax Analysis :

- Syntax analysis imposes a hierarchical structure on the token stream. This hierarchical structure is called a syntax tree.
- A syntax tree has an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
- A leaf is a record with two or more fields, one to identify the token at the leaf, and the other to record information about the token.

3. Semantic Analysis :

- This phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- An important component of semantic analysis is type checking.

Lexical Analysis

We have made a scanner for C Programming Language. The following features have been taken care of :

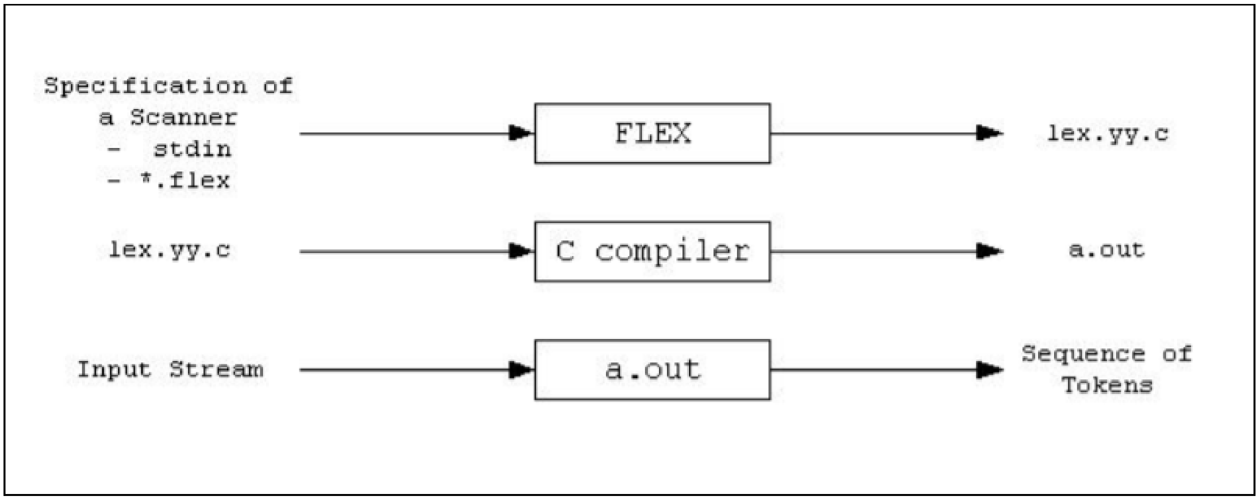
1. Identification of Keywords, Identifiers, Operators (Relational, Logical and Arithmetic), Punctuators, Constants (Integer, Character and Float)
2. Single and Multi Line Comments
3. Data Types (int, float and char) with modifiers (unsigned and signed) and types (short,long).
4. Control statements (IF, IF..ELSE) and Loop statements (FOR, WHILE)

The following lexical errors have been handled :

1. Nested Comments
2. Invalid Identifiers

Flex

FLEX The scanner performs lexical analysis of a certain program. It reads the source program as a sequence of characters and recognizes larger textual units called tokens.
FLEX stands for Fast Lexical Analyzer Generator. It is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

- *.lex is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner.

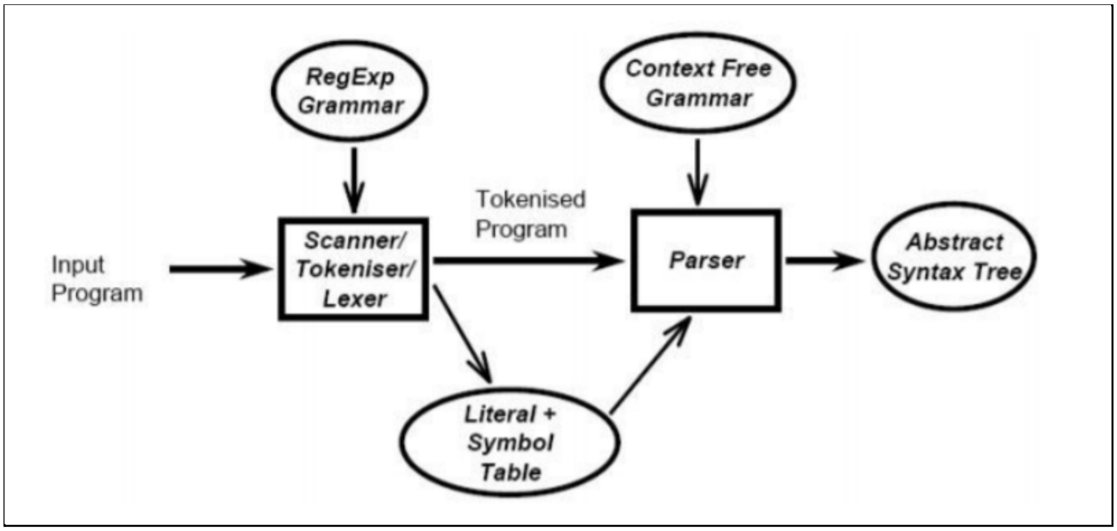
These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right-moving Turing machines. The syntax is based on the use of regular expressions.

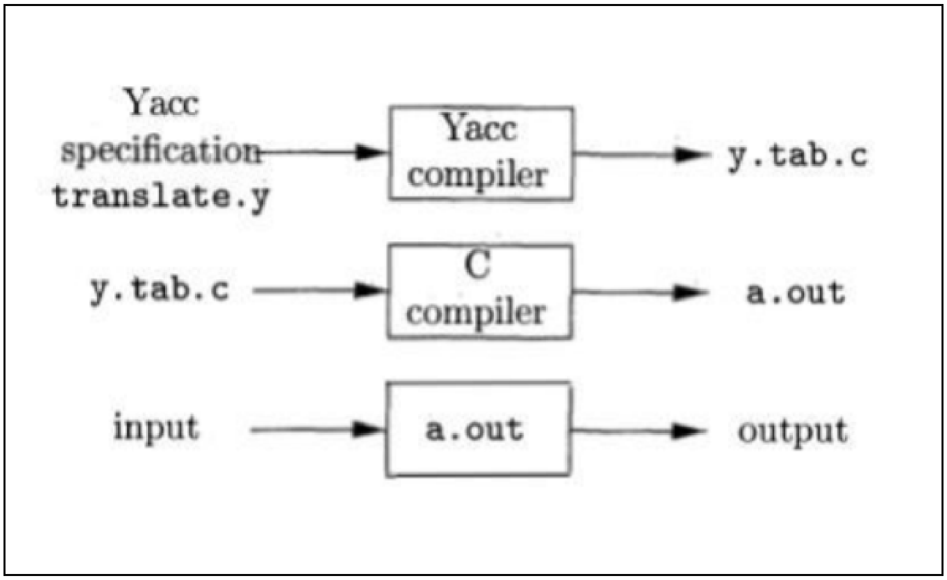
Syntax Analysis

In this second phase of this project, we have implemented and added a Syntactic Analyzer/Parser to the C Compiler. This takes in the stream of tokens generated by the Lexical Analyzer implemented in the first phase as input. For the output, it displays syntax errors along with their corresponding line number. Symbol table and Constant table have also been added.

The following list of features have been included in this phase :

1. Syntax checking for Arithmetic, Logical and Relational Expressions
2. IF, IF...ELSE and Nested IF statements
3. Validation of Unary Operators
4. Validation of all loops - FOR, WHILE, DO...WHILE
5. Missing semicolon at the end of statements
6. Unbalanced Parentheses





Semantic Analysis

In this third phase of the project, we have implemented and added a Semantic Analyzer to the C Compiler. Semantic analysis checks whether the parse tree constructed follows the rules of language. Also, the semantic analyzer keeps track of identifiers, their types and expressions, whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

Semantic analysis typically involves:

1. Type checking - Type mismatch error
2. Multiple declaration with same or different data types error
3. Undeclared variable error
4. Separating variable declarations based on the scope of a variable or the basic block in which a variable is declared.

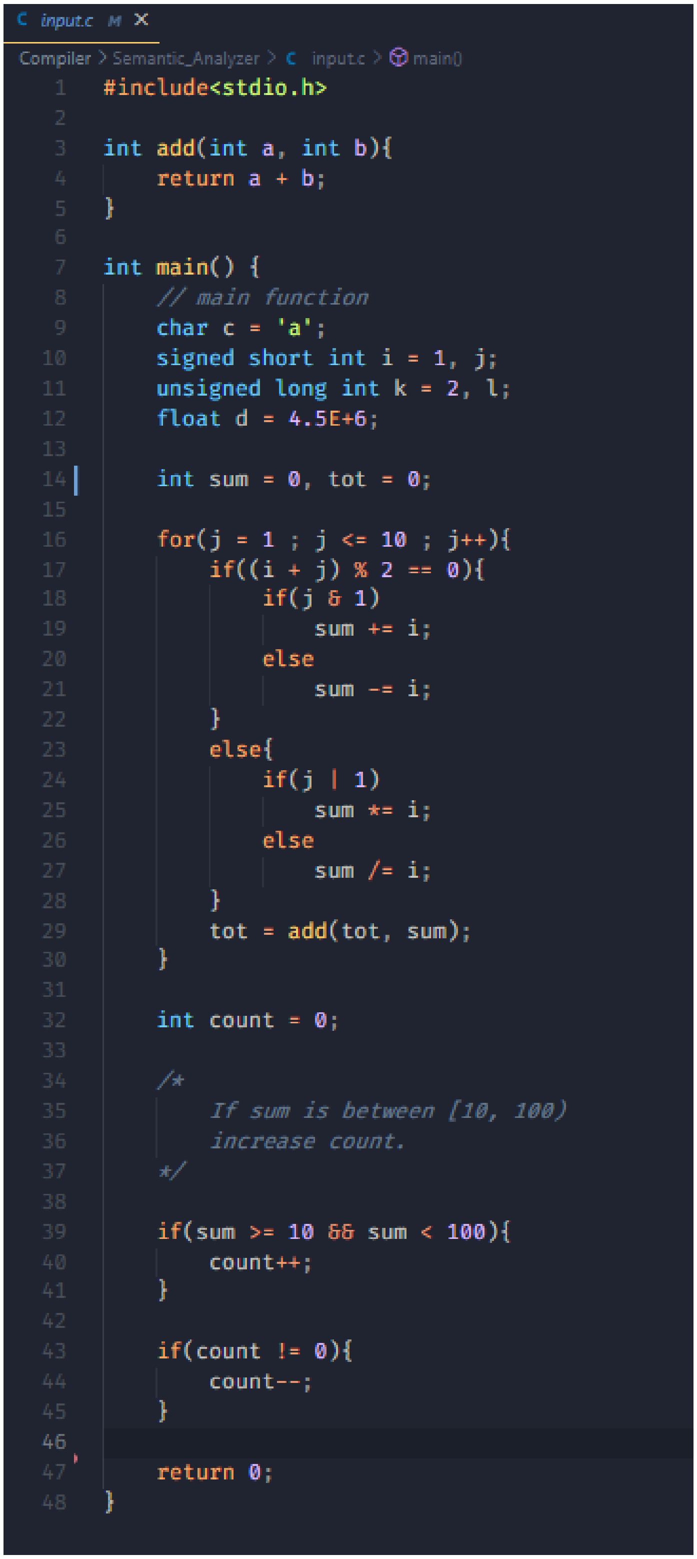
Symbol Table

It contains fields:

1. Name : identifier name.
2. Token : token is constant or identifier.
3. Type : type of identifier whether int, float, void etc.
4. Scope : Scope could be global, function.
5. Scope-id : Unique value given to each block of code.
6. Line Numbers: List of line numbers where a token is defined/used.

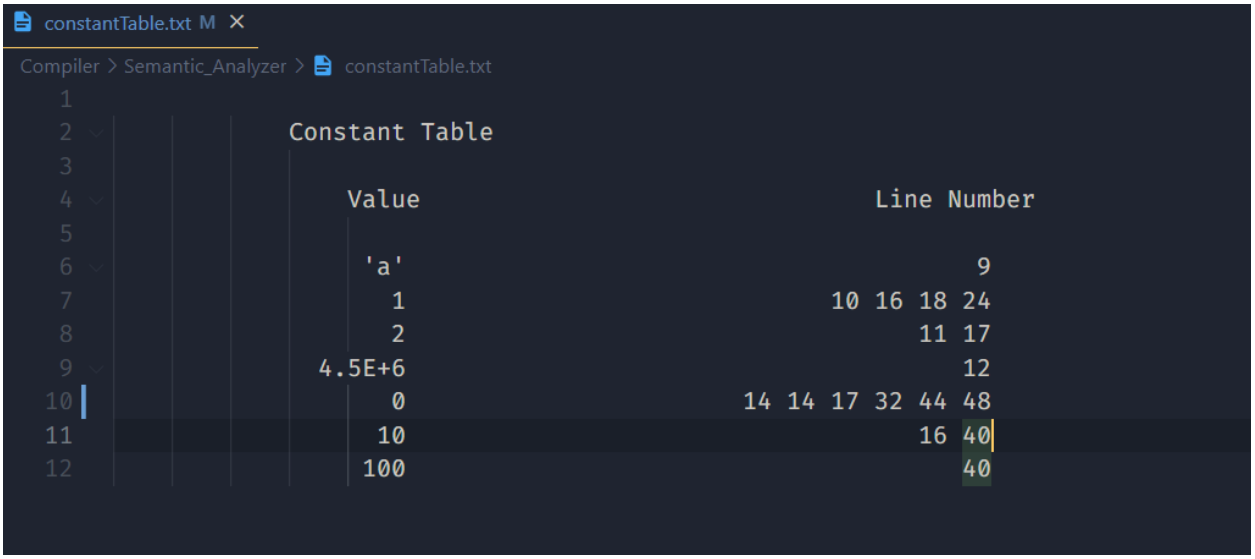
A sample program and the information stored in Constant Table, Parsed Table and Symbol Table:

Input



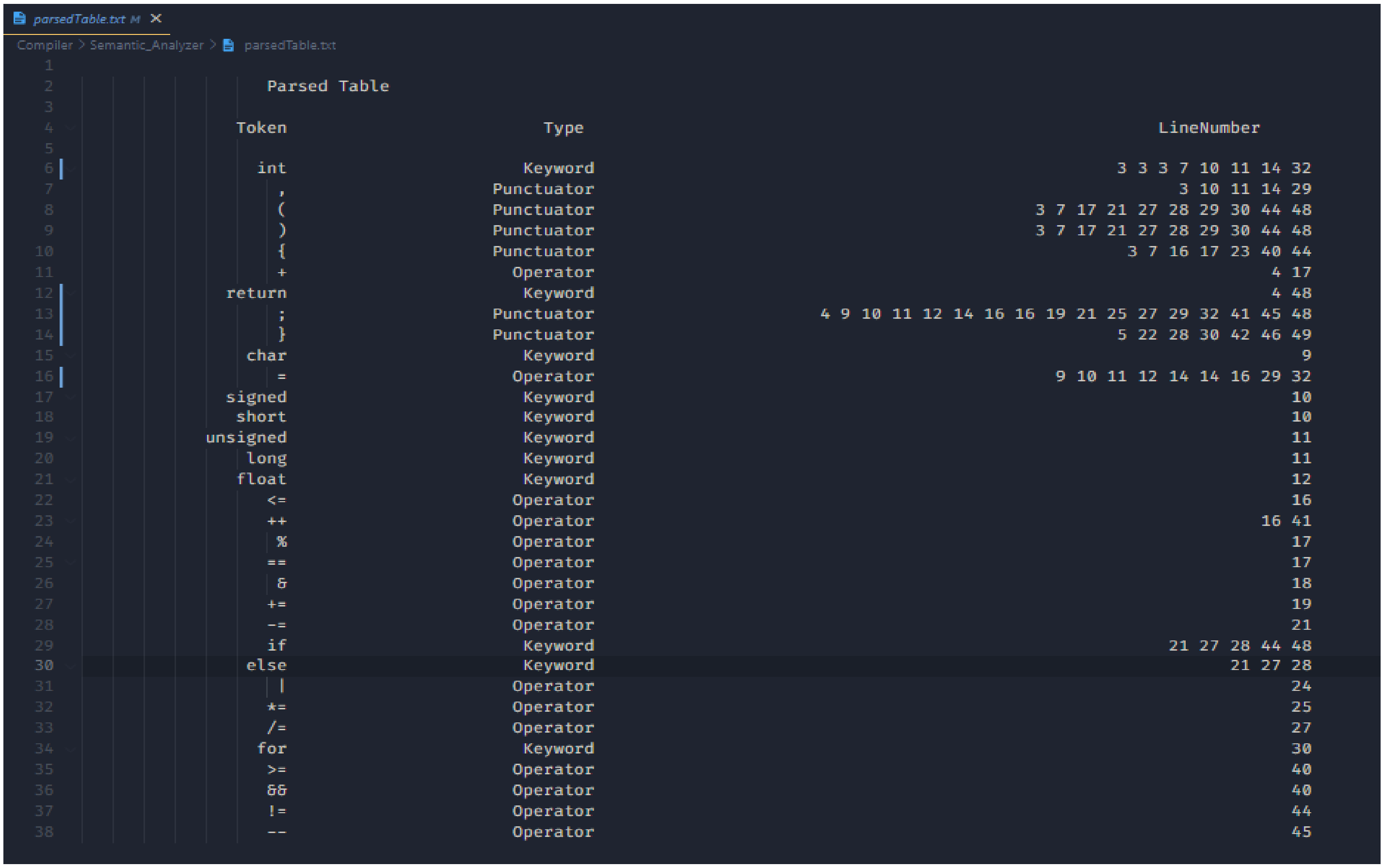
```c
#include<stdio.h>

int add(int a, int b){
    return a + b;
}

int main() {
    // main function
    char c = 'a';
    signed short int i = 1, j;
    unsigned long int k = 2, l;
    float d = 4.5E+6;

    int sum = 0, tot = 0;

    for(j = 1 ; j <= 10 ; j++){
        if((i + j) % 2 == 0){
            if(j & 1)
                sum += i;
            else
                sum -= i;
        }
        else{
            if(j | 1)
                sum *= i;
            else
                sum /= i;
        }
        tot = add(tot, sum);
    }

    int count = 0;

    /*
        If sum is between [10, 100)
        increase count.
    */

    if(sum >= 10 && sum < 100){
        count++;
    }

    if(count != 0){
        count--;
    }

    return 0;
}
```

Constant Table



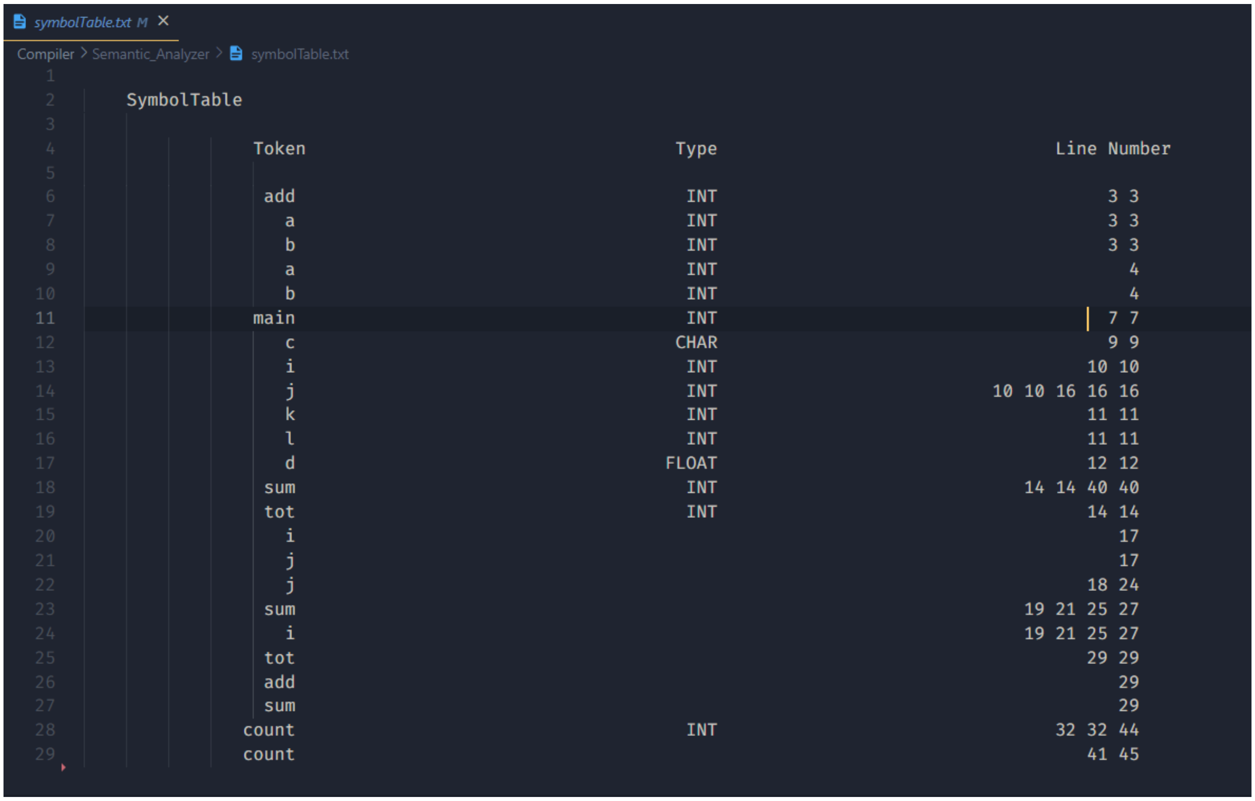
```
                Constant Table

                   Value                       Line Number

                    'a'                                  9
                      1                        10 16 18 24
                      2                              11 17
                 4.5E+6                                 12
                      0                  14 14 17 32 44 48
                     10                              16 40
                    100                                 40
```

Parsed Table



```
            Parsed Table

        Token                     Type                                          LineNumber

          int                  Keyword                                 3 3 3 7 10 11 14 32
            ,               Punctuator                                     3 10 11 14 29
            (               Punctuator                        3 7 17 21 27 28 29 30 44 48
            )               Punctuator                        3 7 17 21 27 28 29 30 44 48
            {               Punctuator                               3 7 16 17 23 40 44
            +                 Operator                                              4 17
       return                  Keyword                                              4 48
            ;               Punctuator     4 9 10 11 12 14 16 16 19 21 25 27 29 32 41 45 48
            }               Punctuator                                5 22 28 30 42 46 49
         char                  Keyword                                                 9
            =                 Operator                         9 10 11 12 14 14 16 29 32
       signed                  Keyword                                                10
        short                  Keyword                                                10
     unsigned                  Keyword                                                11
         long                  Keyword                                                11
        float                  Keyword                                                12
           <=                 Operator                                                16
           ++                 Operator                                             16 41
            %                 Operator                                                17
           ==                 Operator                                                17
            &                 Operator                                                18
           +=                 Operator                                                19
           -=                 Operator                                                21
           if                  Keyword                                    21 27 28 44 48
         else                  Keyword                                          21 27 28
            |                 Operator                                                24
           *=                 Operator                                                25
           /=                 Operator                                                27
          for                  Keyword                                                30
           >=                 Operator                                                40
           &&                 Operator                                                40
           !=                 Operator                                                44
           --                 Operator                                                45
```

Symbol Table



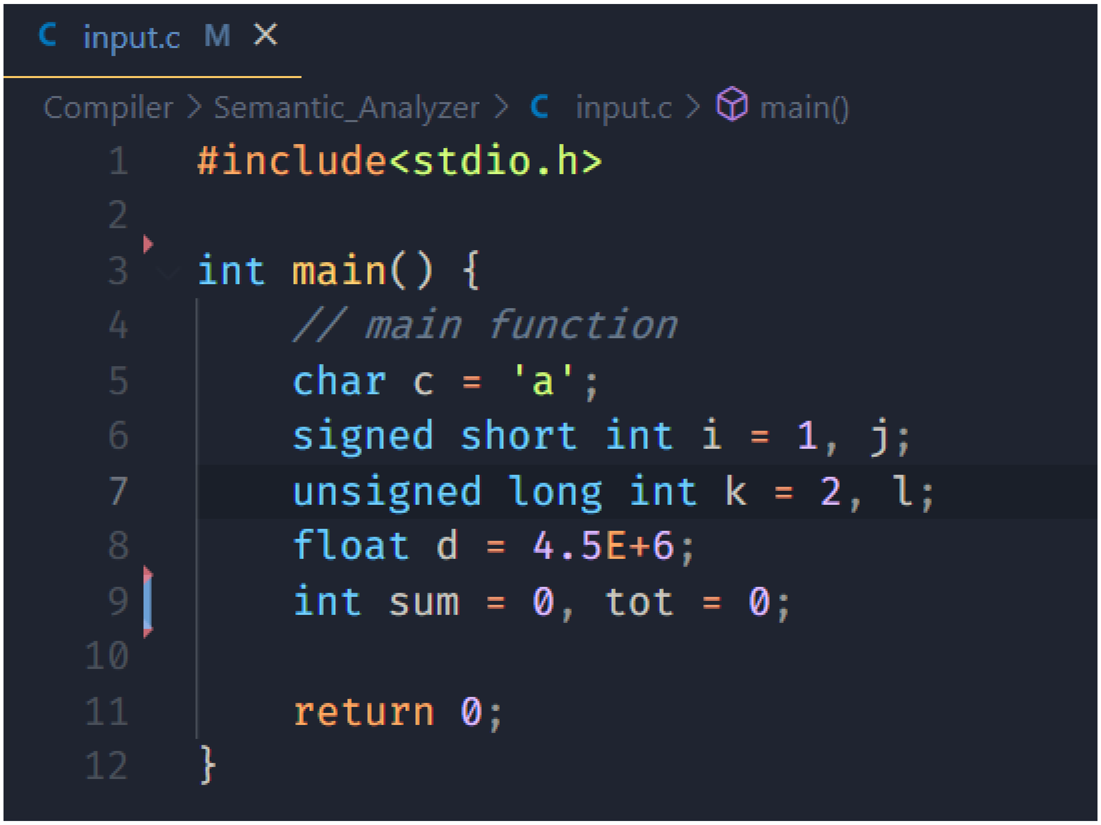
```

    SymbolTable

                Token                                   Type                                Line Number

                 add                                     INT                                     3 3
                   a                                     INT                                     3 3
                   b                                     INT                                     3 3
                   a                                     INT                                       4
                   b                                     INT                                       4
                main                                     INT                                     7 7
                   c                                    CHAR                                     9 9
                   i                                     INT                                   10 10
                   j                                     INT                          10 10 16 16 16
                   k                                     INT                                   11 11
                   l                                     INT                                   11 11
                   d                                   FLOAT                                   12 12
                 sum                                     INT                             14 14 40 40
                 tot                                     INT                                   14 14
                   i                                                                              17
                   j                                                                              17
                   j                                                                           18 24
                 sum                                                                     19 21 25 27
                   i                                                                     19 21 25 27
                 tot                                                                           29 29
                 add                                                                              29
                 sum                                                                              29
               count                                     INT                                32 32 44
               count                                                                           41 45
```

Test Cases

Case 1

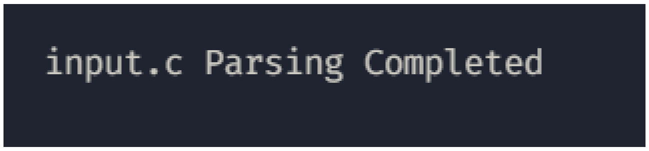
Purpose: Test case to check int, char and float constants, keywords, punctuators, variables and type specifiers.

Input



```c
#include<stdio.h>

int main() {
    // main function
    char c = 'a';
    signed short int i = 1, j;
    unsigned long int k = 2, l;
    float d = 4.5E+6;
    int sum = 0, tot = 0;

    return 0;
}
```

Output

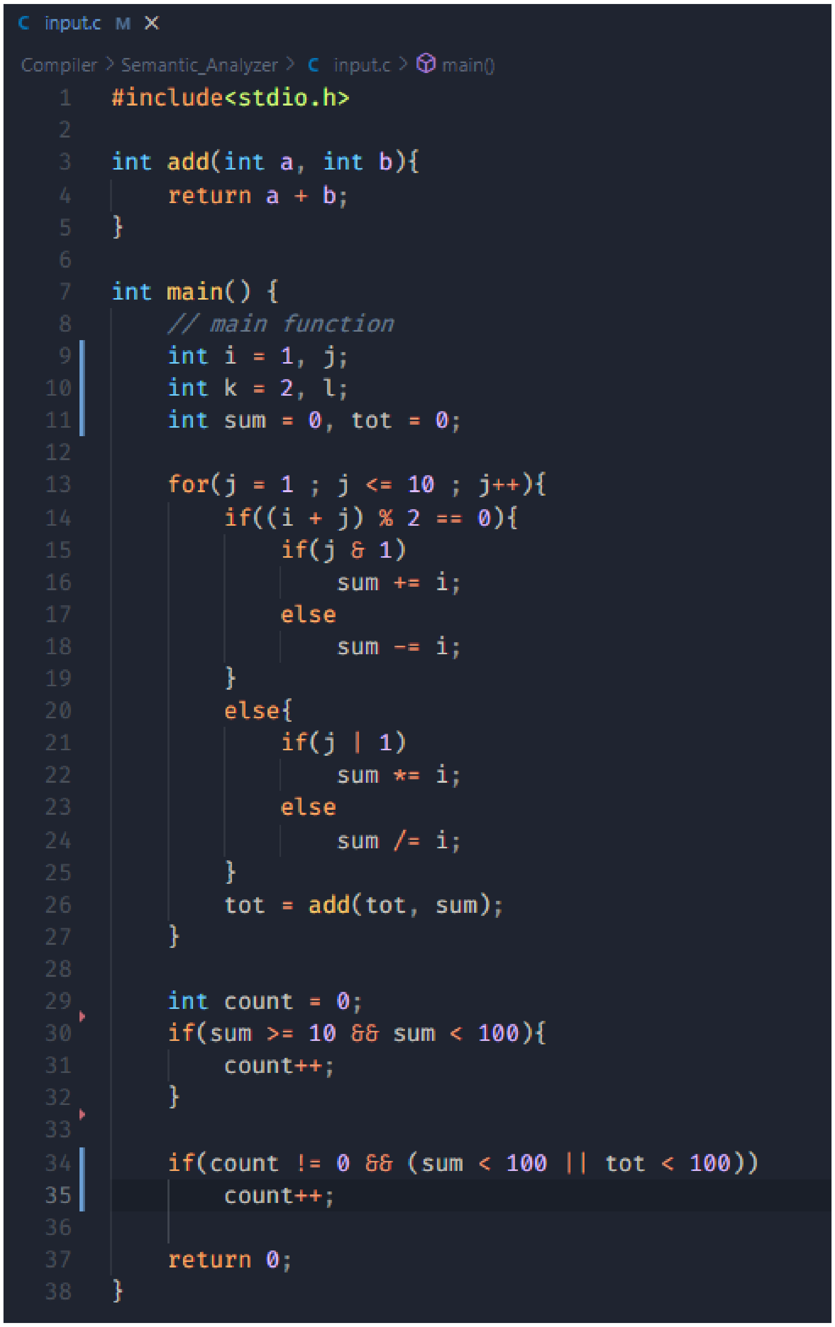


```
input.c Parsing Completed
```

Case 2

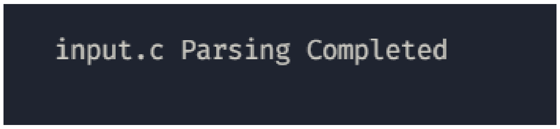
Purpose: Test case to check relational, logical and arithmetic operators

Input



```c
#include<stdio.h>

int add(int a, int b){
    return a + b;
}

int main() {
    // main function
    int i = 1, j;
    int k = 2, l;
    int sum = 0, tot = 0;

    for(j = 1 ; j <= 10 ; j++){
        if((i + j) % 2 == 0){
            if(j & 1)
                sum += i;
            else
                sum -= i;
        }
        else{
            if(j | 1)
                sum *= i;
            else
                sum /= i;
        }
        tot = add(tot, sum);
    }

    int count = 0;
    if(sum >= 10 && sum < 100){
        count++;
    }

    if(count != 0 && (sum < 100 || tot < 100))
        count++;

    return 0;
}
```

Output

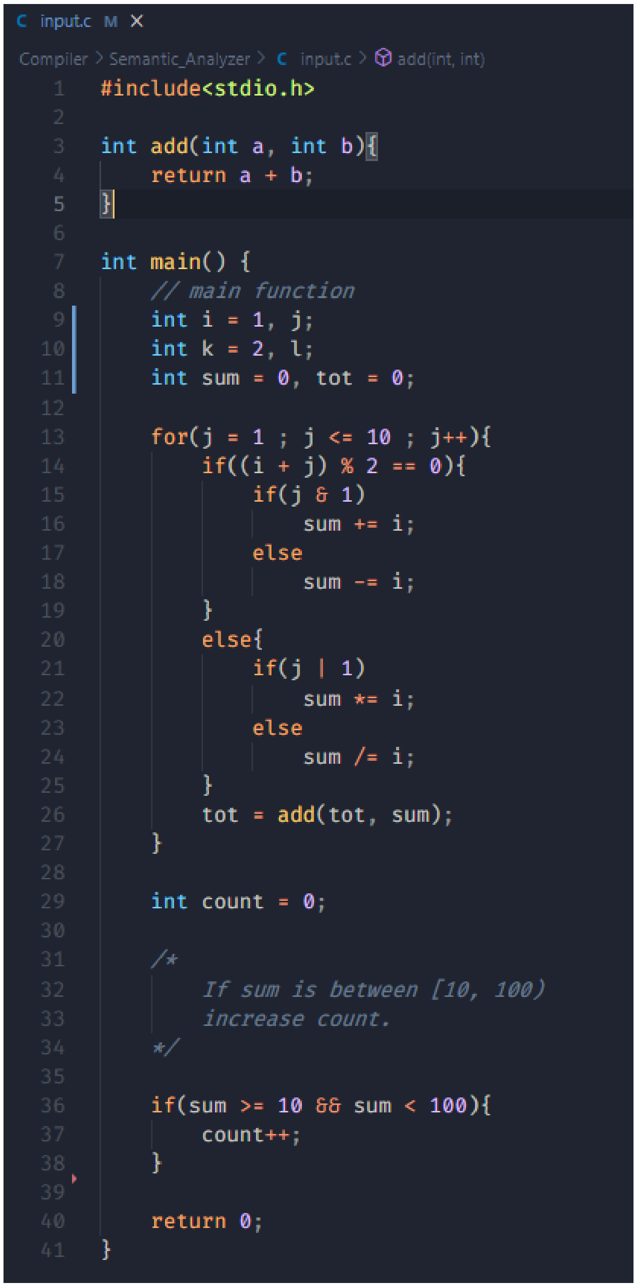


```
input.c Parsing Completed
```

Case 3

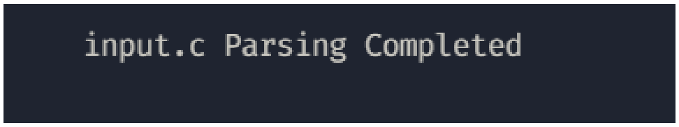
Purpose: Test case to check single and multi line comments

Input



```c
#include<stdio.h>

int add(int a, int b){
    return a + b;
}

int main() {
    // main function
    int i = 1, j;
    int k = 2, l;
    int sum = 0, tot = 0;

    for(j = 1 ; j <= 10 ; j++){
        if((i + j) % 2 == 0){
            if(j & 1)
                sum += i;
            else
                sum -= i;
        }
        else{
            if(j | 1)
                sum *= i;
            else
                sum /= i;
        }
        tot = add(tot, sum);
    }

    int count = 0;

    /*
        If sum is between [10, 100)
        increase count.
    */

    if(sum >= 10 && sum < 100){
        count++;
    }

    return 0;
}
```

Output

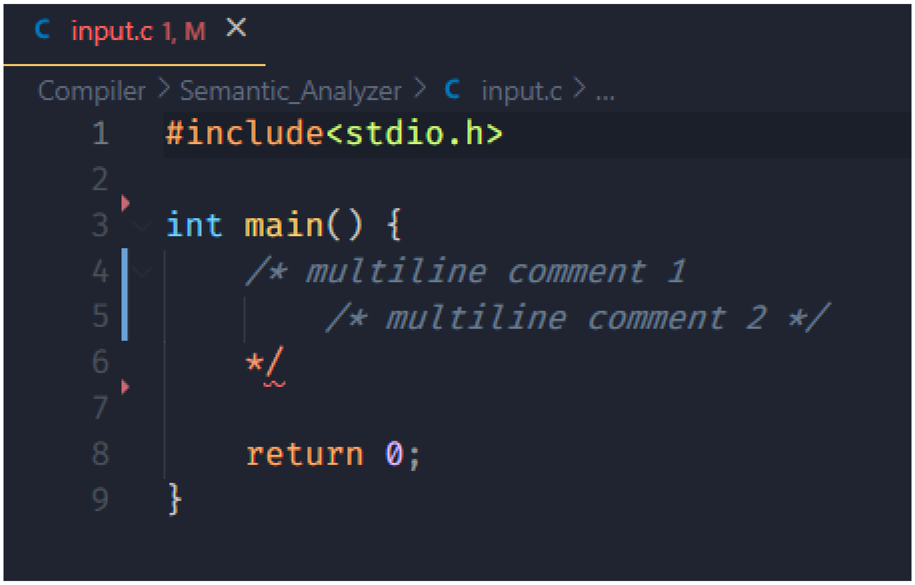


```
input.c Parsing Completed
```

Case 4

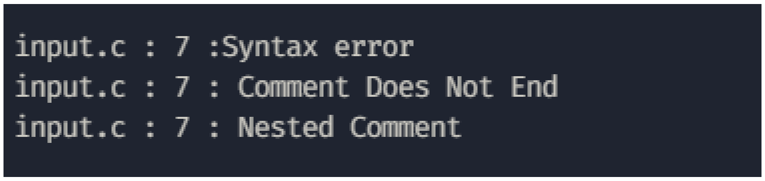
Purpose: Test case to check nested multiline comment error

Input



```c
#include<stdio.h>

int main() {
    /* multiline comment 1
        /* multiline comment 2 */
    */

    return 0;
}
```

Output

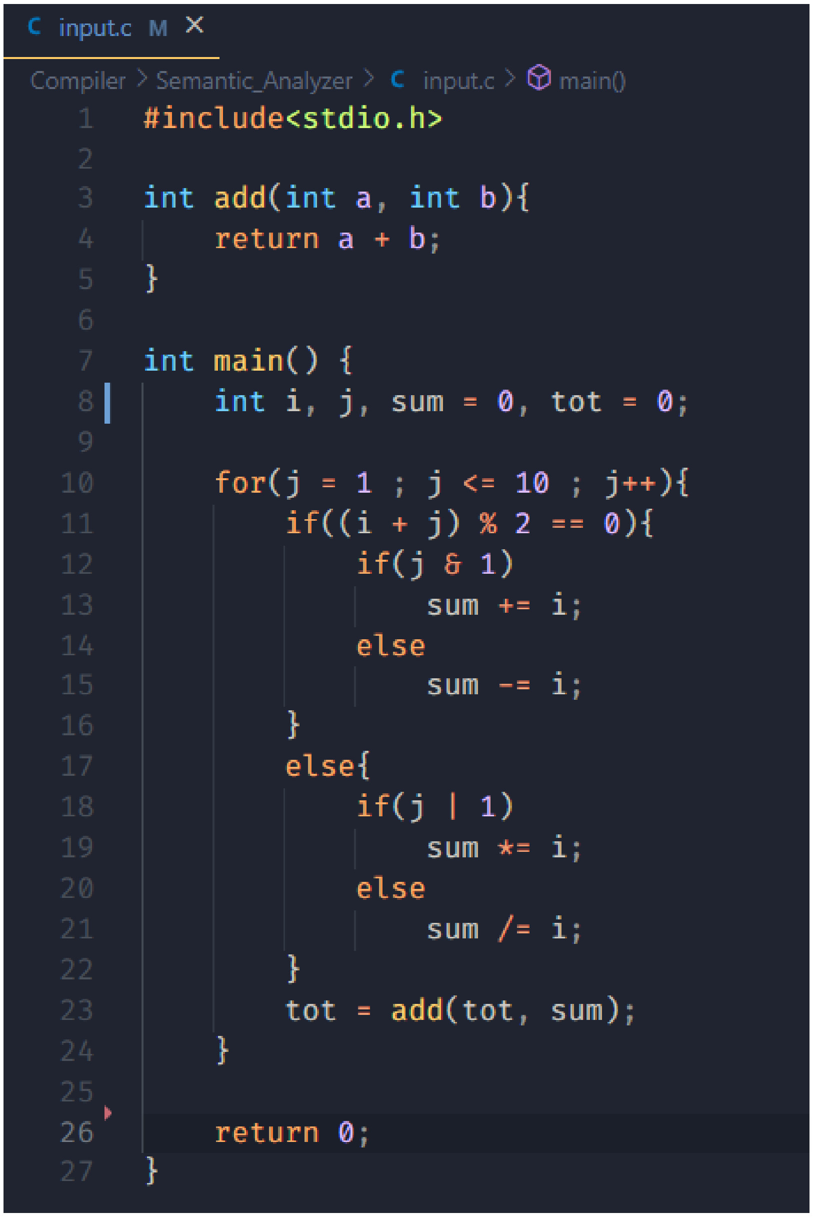


```
input.c : 7 :Syntax error
input.c : 7 : Comment Does Not End
input.c : 7 : Nested Comment
```

Case 5

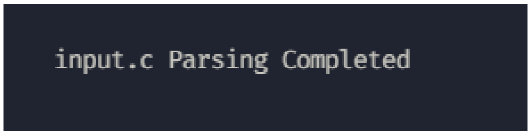
Purpose: Test case to check control and looping statements

Input



```c
#include<stdio.h>

int add(int a, int b){
    return a + b;
}

int main() {
    int i, j, sum = 0, tot = 0;

    for(j = 1 ; j <= 10 ; j++){
        if((i + j) % 2 == 0){
            if(j & 1)
                sum += i;
            else
                sum -= i;
        }
        else{
            if(j | 1)
                sum *= i;
            else
                sum /= i;
        }
        tot = add(tot, sum);
    }

    return 0;
}
```

Output

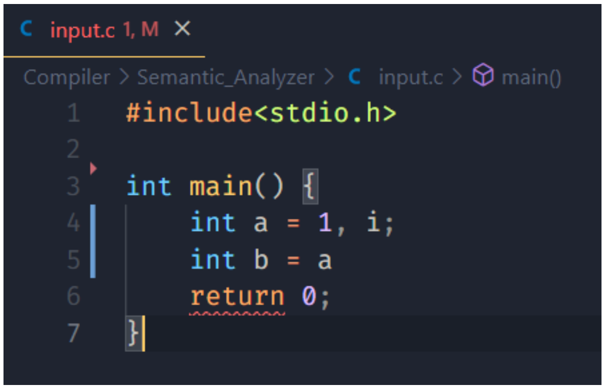


```
input.c Parsing Completed
```

Case 6

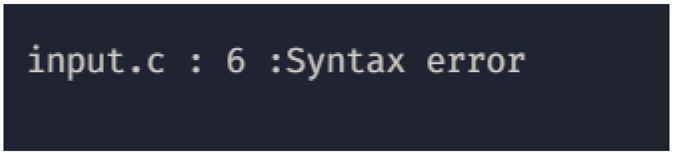
Purpose: Test case to check missing semicolon, unbalanced parentheses

Input



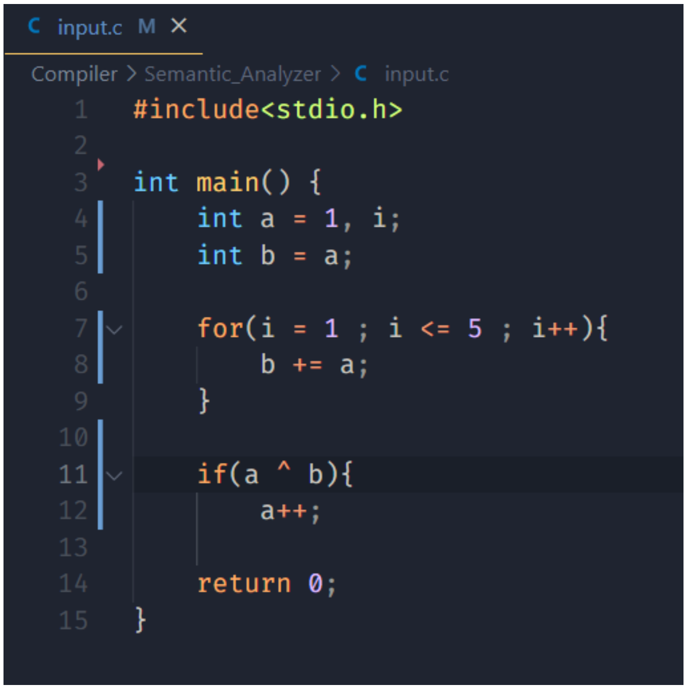
```c
#include<stdio.h>

int main() {
    int a = 1, i;
    int b = a
    return 0;
}
```

Output



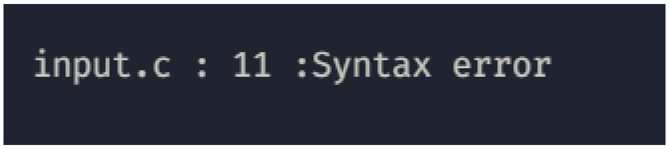
```
input.c : 6 :Syntax error
```

Input



```c
#include<stdio.h>

int main() {
    int a = 1, i;
    int b = a;

    for(i = 1 ; i <= 5 ; i++){
        b += a;
    }

    if(a ^ b){
        a++;

    return 0;
}
```

Output

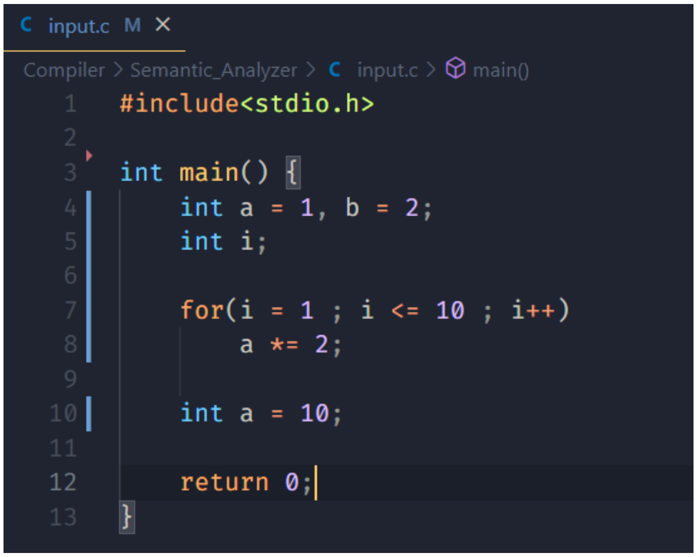


```
input.c : 11 :Syntax error
```

Case 7

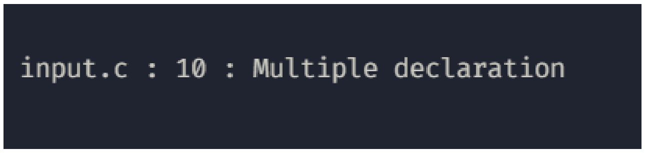
Purpose: Test case to check multiple declarations

Input



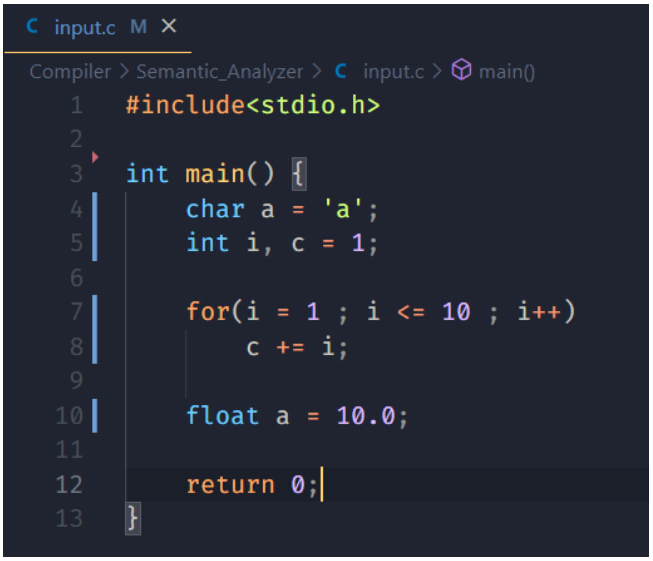
```c
#include<stdio.h>

int main() {
    int a = 1, b = 2;
    int i;

    for(i = 1 ; i <= 10 ; i++)
        a *= 2;

    int a = 10;

    return 0;
}
```

Output



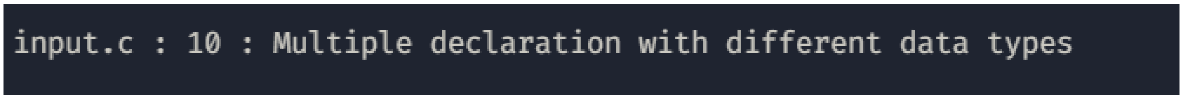
```
input.c : 10 : Multiple declaration
```

Input



```c
#include<stdio.h>

int main() {
    char a = 'a';
    int i, c = 1;

    for(i = 1 ; i <= 10 ; i++)
        c += i;

    float a = 10.0;

    return 0;
}
```

Output

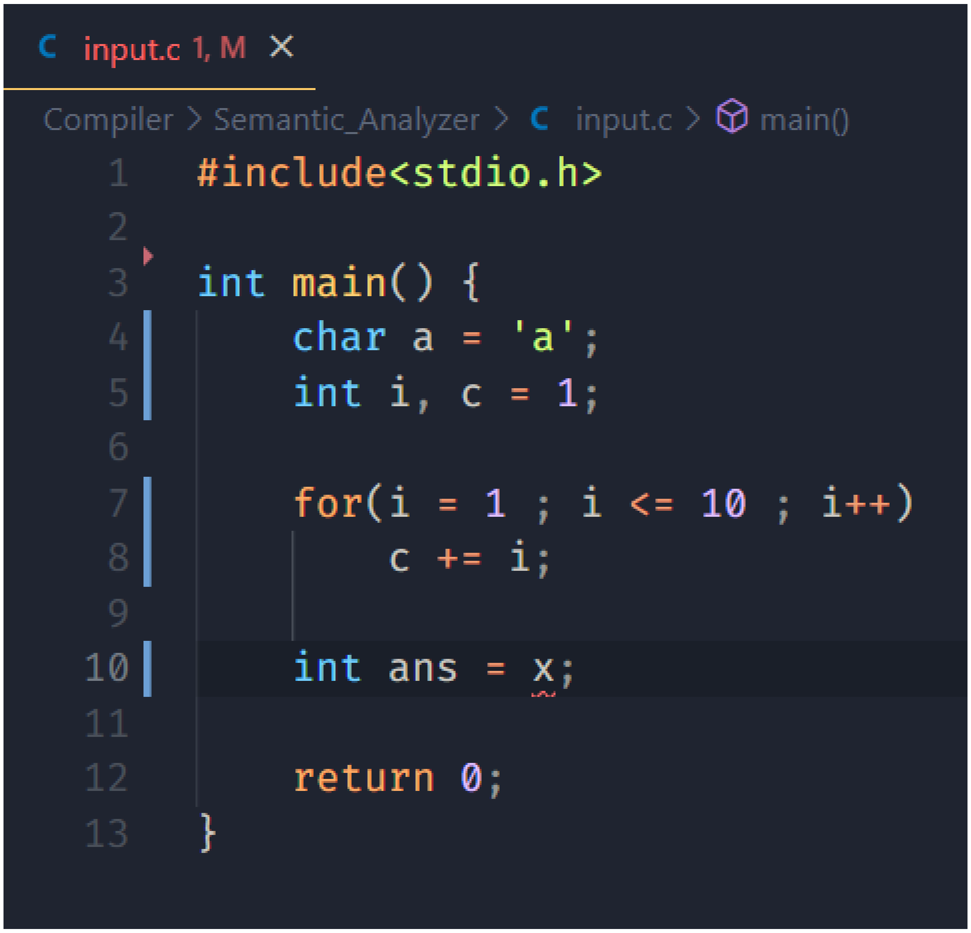


```
input.c : 10 : Multiple declaration with different data types
```

Case 8

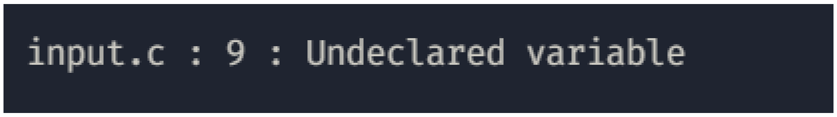
Purpose: Test case to check undeclared variables

Input



```c
#include<stdio.h>

int main() {
    char a = 'a';
    int i, c = 1;

    for(i = 1 ; i <= 10 ; i++)
        c += i;

    int ans = x;

    return 0;
}
```

Output

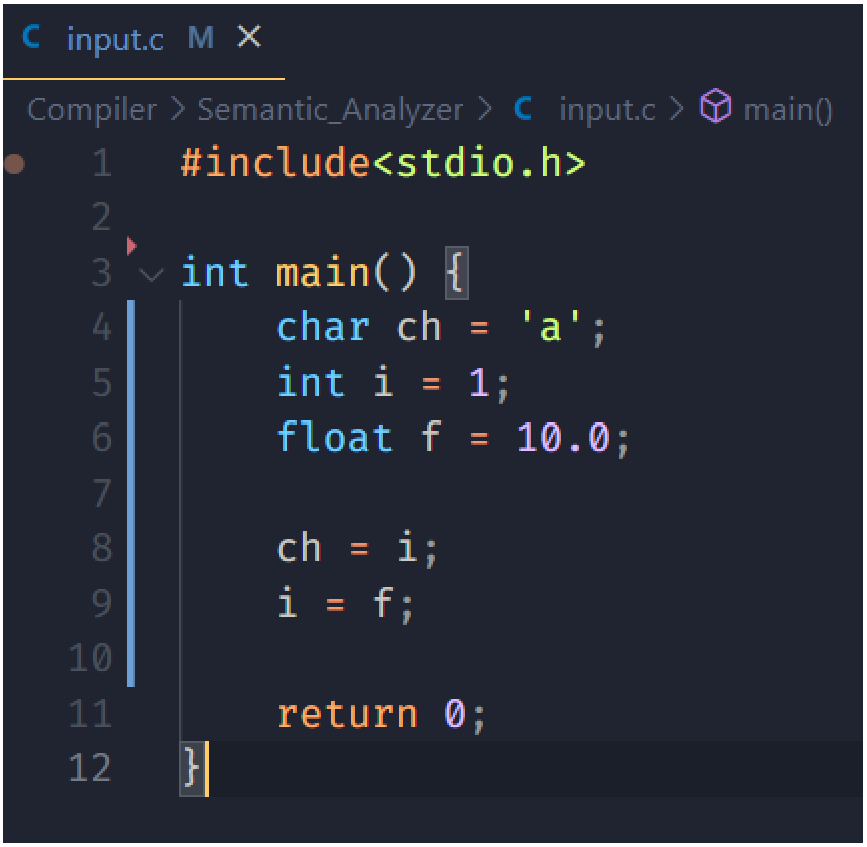


```
input.c : 9 : Undeclared variable
```

Case 9

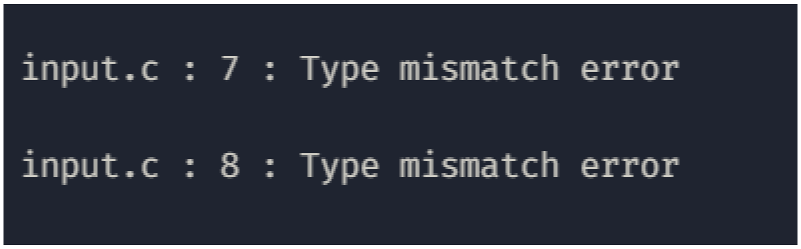
Purpose: Test case to check type mismatch

Input



```
1   #include<stdio.h>
2
3   int main() {
4       char ch = 'a';
5       int i = 1;
6       float f = 10.0;
7
8       ch = i;
9       i = f;
10
11      return 0;
12  }
```

Output

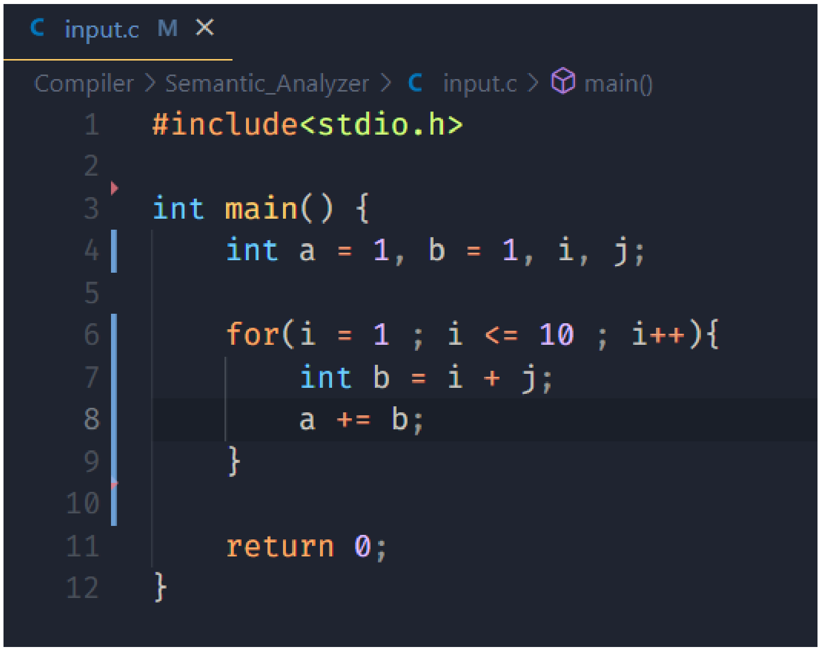


```
input.c : 7 : Type mismatch error

input.c : 8 : Type mismatch error
```

Case 10

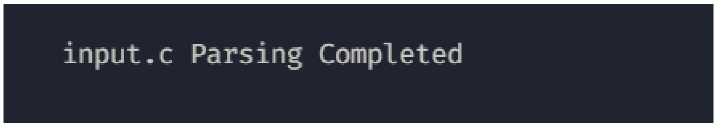
Purpose: Test case to check scope of a variable

Input



```c
#include<stdio.h>

int main() {
    int a = 1, b = 1, i, j;

    for(i = 1 ; i <= 10 ; i++){
        int b = i + j;
        a += b;
    }

    return 0;
}
```

Output



```
input.c Parsing Completed
```

User Guide

From /Compiler/Semantic_Analyzer directory run the following commands or bash script run.sh. Here, input.c contains the input program.

```
bison -d compiler.y
flex compiler.l
gcc compiler.tab.c lex.yy.c
./a.exe input.c
rm compiler.tab.c compiler.tab.h lex.yy.c
```

Click here to see the Code.