

**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.

1. **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.

1. **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)

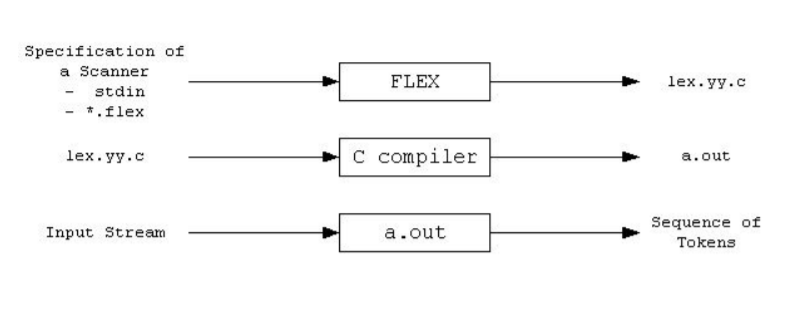
The following lexical errors have been handled :

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.

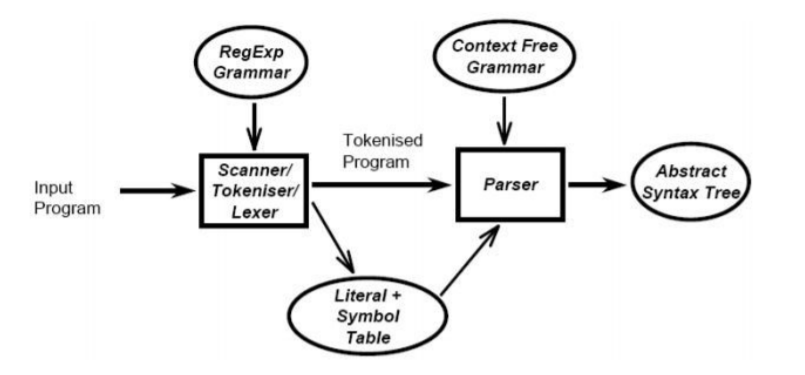
These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right-moving Turing machines. The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

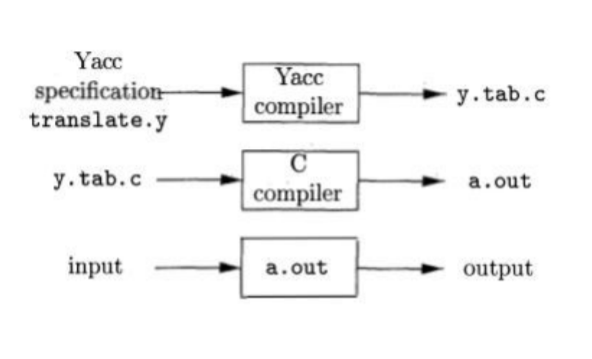
**Syntax Analysis**

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.

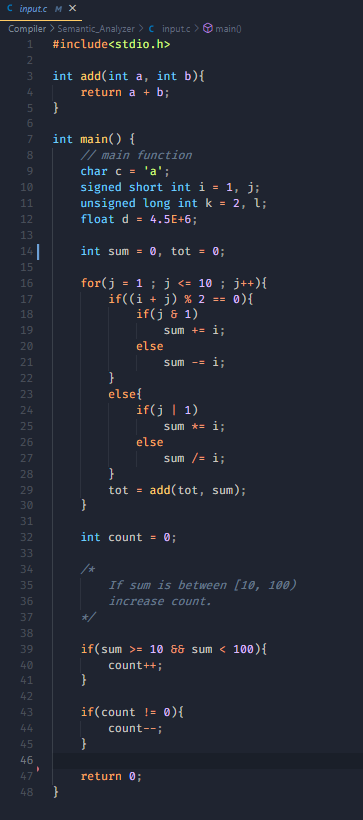
**Symbol Table**

It contains fields:

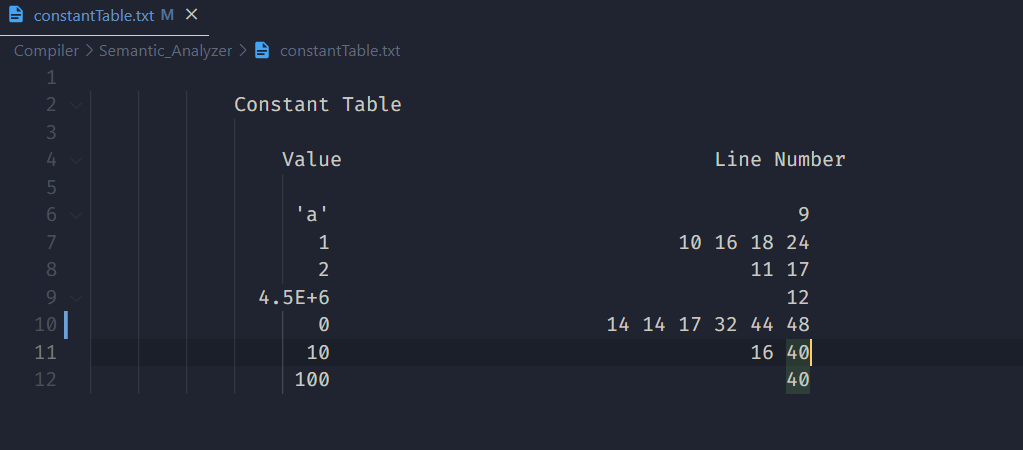
1. Name : identifier name.
2. Token : token is constant or identifier.
3. Type : type of identifier whether int, float, void etc.
4. Scope : Scope could be global, function.
5. Scope-id : Unique value given to each block of code.
6. Line Numbers: List of line numbers where a token is defined/used.

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

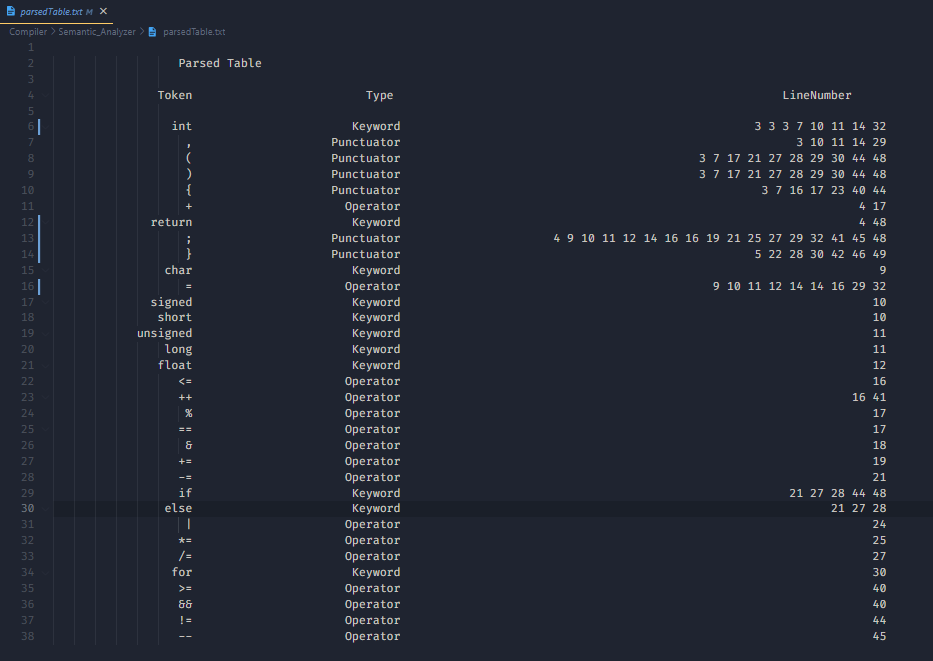
**Input**

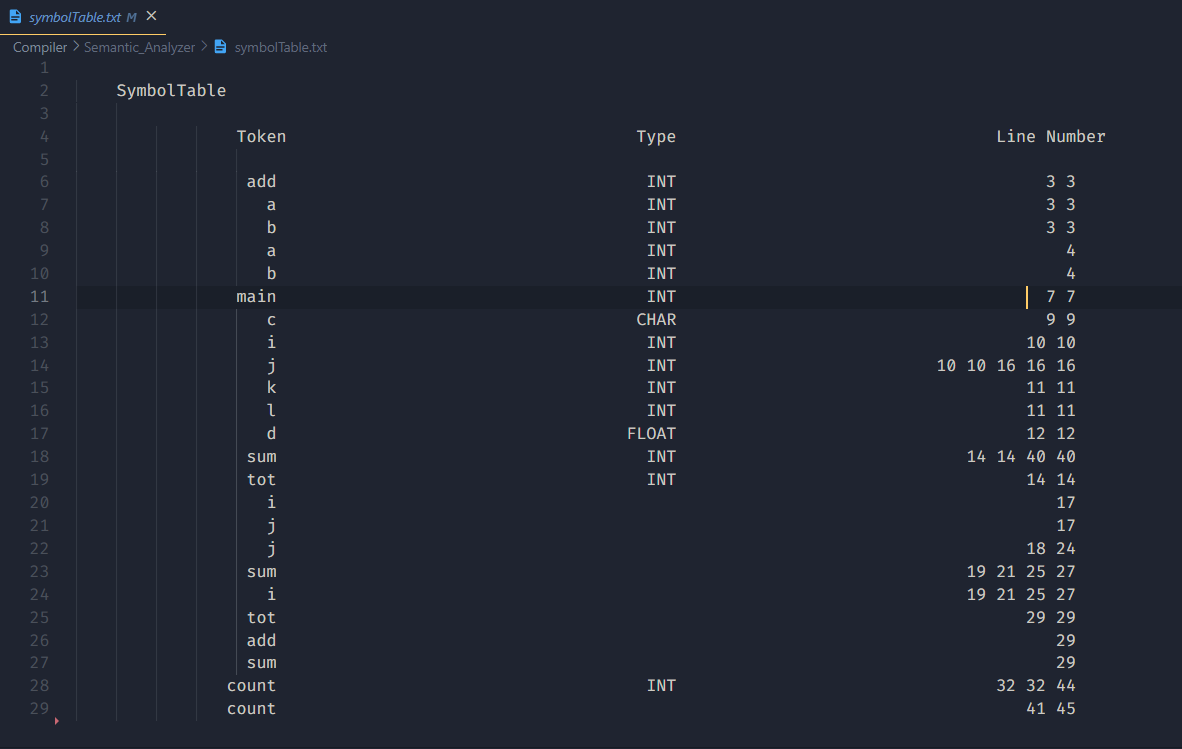


**Constant Table**



**Parsed Table**



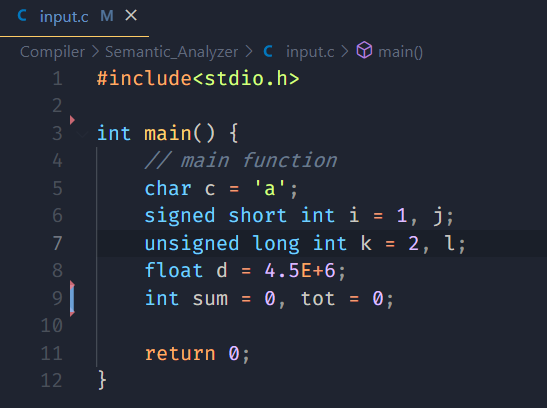
**Symbol Table**

**Test Cases**

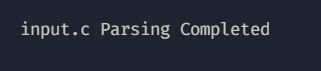
**Case 1**

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

**Input**

****

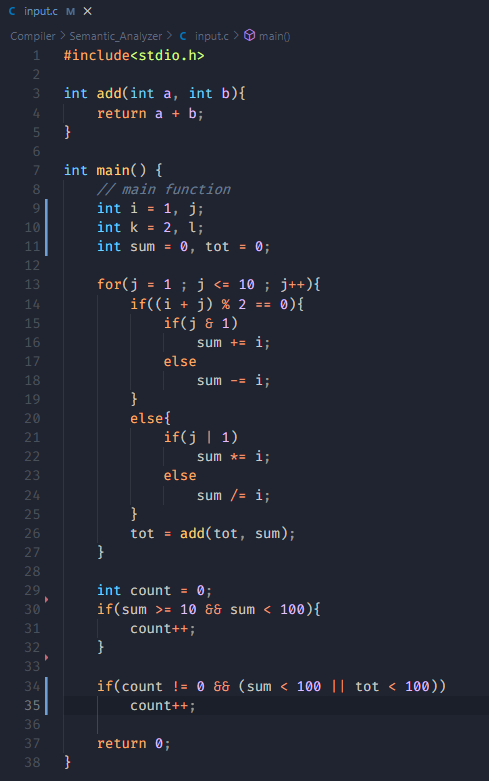
**Output**

****

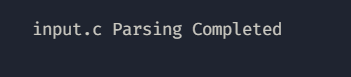
**Case 2**

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

**Input**

****

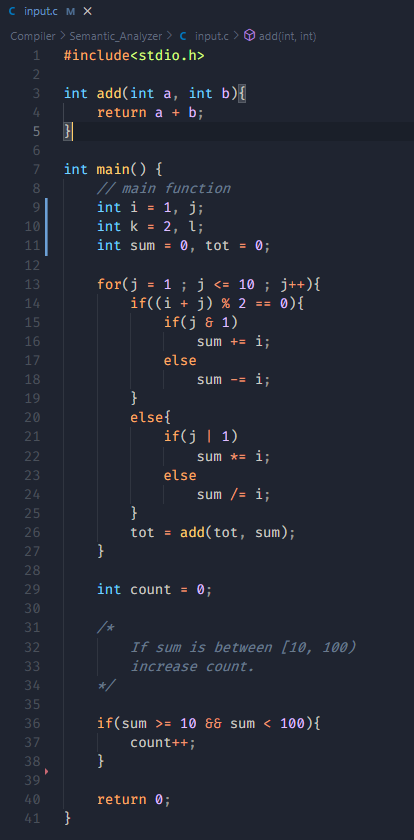
**Output**

****

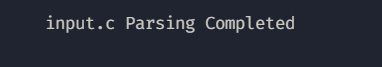
**Case 3**

**Purpose: Test case to check single and multi line comments**

**Input**

****

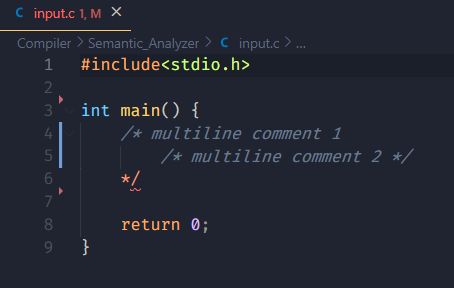
**Output**

****

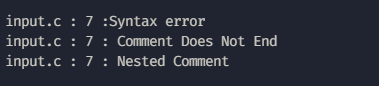
**Case 4**

**Purpose: Test case to check nested multiline comment error**

**Input**

****

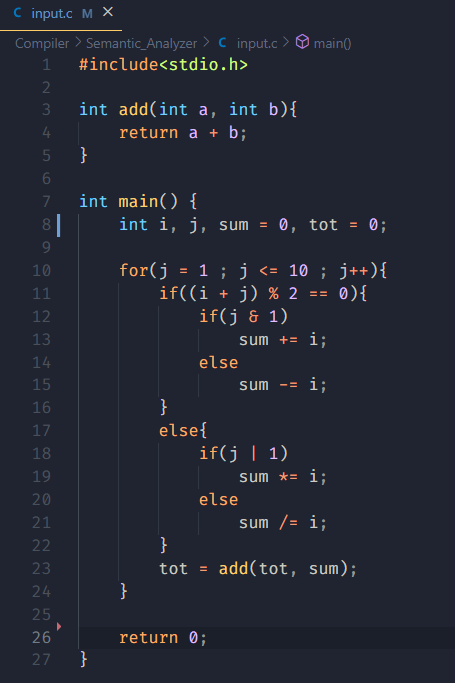
**Output**

****

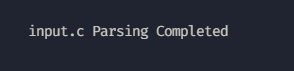
**Case 5**

**Purpose: Test case to check control and looping statements**

**Input**

****

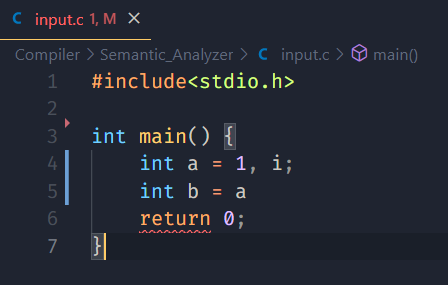
**Output**

****

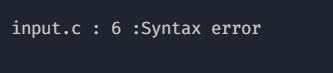
**Case 6**

**Purpose: Test case to check missing semicolon, unbalanced parentheses**

**Input**

****

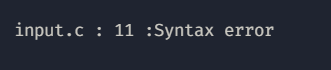
**Output**

****

**Input**

****

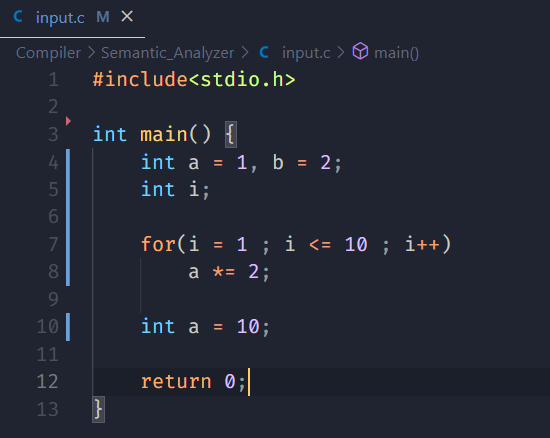
**Output**

****

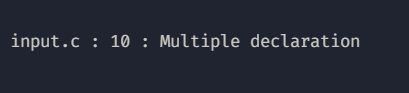
**Case 7**

**Purpose: Test case to check multiple declarations**

**Input**

****

**Output**

****

**Input**

****

**Output**

****

**Case 8**

**Purpose: Test case to check undeclared variables**

**Input**

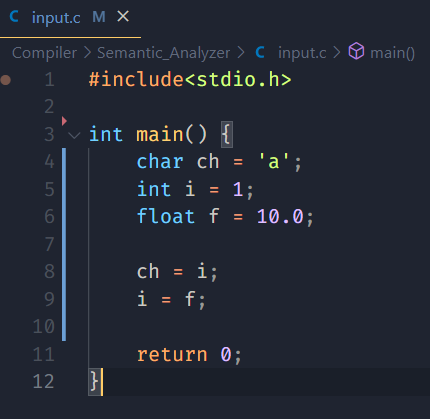
****

**Output**

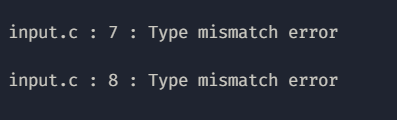
****

**Case 9**

**Purpose: Test case to check type mismatch**

**Input  
**

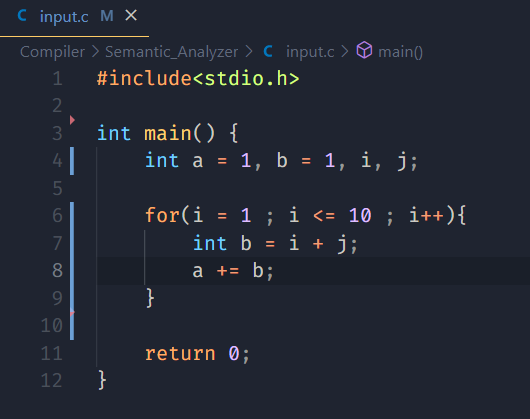
**Output**

****

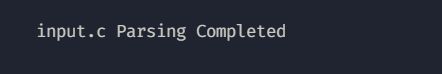
**Case 10**

**Purpose: Test case to check scope of a variable**

**Input**

****

**Output**

****

**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