COMPILER LAB PROJECT-1

LEXICAL ANALYZER USING FLEX

Submitted By:

Sai Priya (15CO116)

S. Venkataraman (15CO140)

Introduction

A compiler, in general, is a computer program that transforms source code written in a programming language into another computer language. The source code is typically in a high-level programming language (e. g. Pascal, C, C++, Java, Perl, C#, etc.). The executable code may be a sequence of machine instructions that can be executed by the CPU directly, or it may be an intermediate representation that is interpreted by a virtual machine (e. g. Java byte code). A compiler has the following main parts

* Lexical Analyzer,
* Syntax Analysis
* Semantic checker,
* Intermediate-code generator,
* Code optimizer and
* Code generator.

Phases of a compiler

**Lexical** **Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

**<token-name,** **attribute-value>**

**Syntax** **Analysis**

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

**Semantic** **Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. 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.

**Intermediate** **Code** **Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**Code** **Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

**Code** **Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

**Symbol** **Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Lexical Analysis

The Lexical analyzer of the compiler is the part which identifies each entry in the source program and differentiate them into lexemes of different tokens. It uses regular expression for achieving this task. Apart from identifying the token it also reports a few lexical errors.

The main task of Lexical Analyzer is to read a stream of characters as an input and produce a sequence of tokens such as names, keywords, punctuation marks etc. for syntax analyzer. It discards the whitespaces and comments between the tokens . It reads a lexeme and identifies which token it belongs to. It also makes an entry into the symbol table

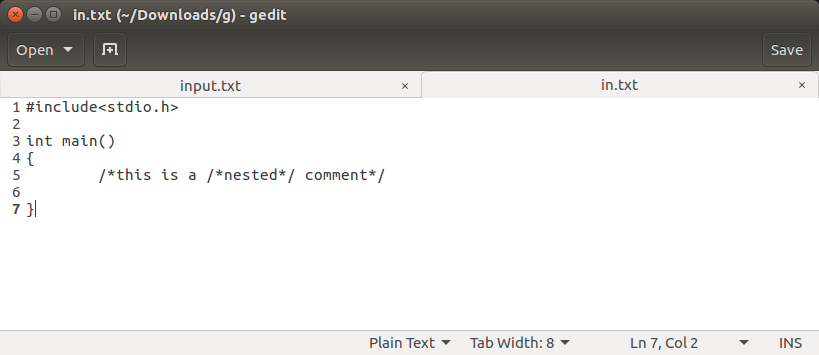
This project makes a Lexical analyzer for a subset of C language using the flex tool. The flex tool allows users to create a scanner, with C language as the base and provides the provision for specifying the regular expression associated with each token.

Lexical Analyzer also generates errors in the following cases:

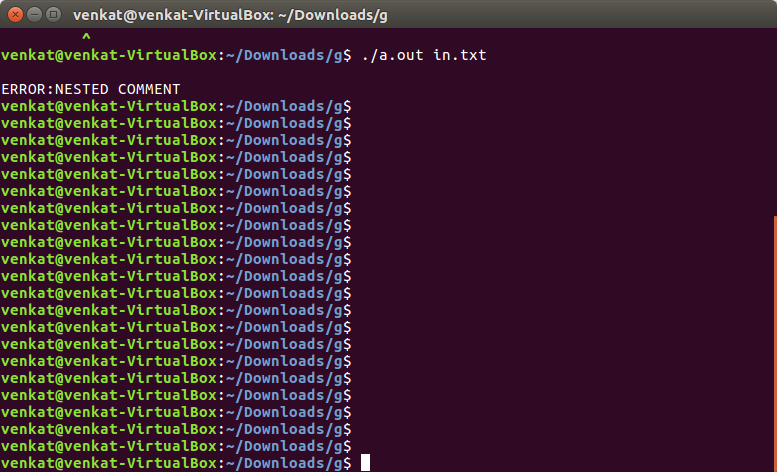
* **Unidentified** **token:** When the lexeme does not match any of the specified regular expressions.
* **Nested** **Comments:** Nested comments are not supported.

Screenshots

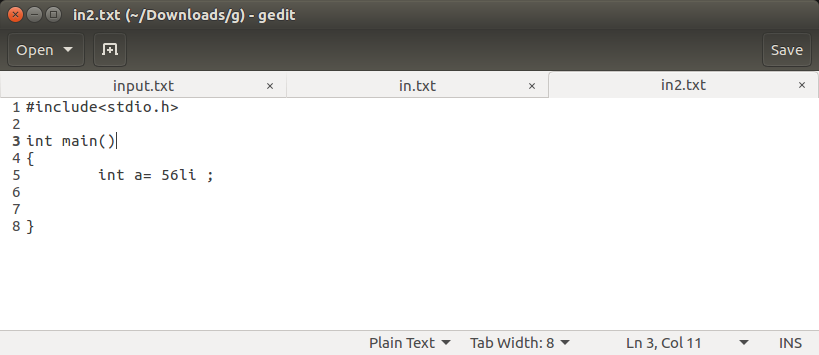
Test case 1



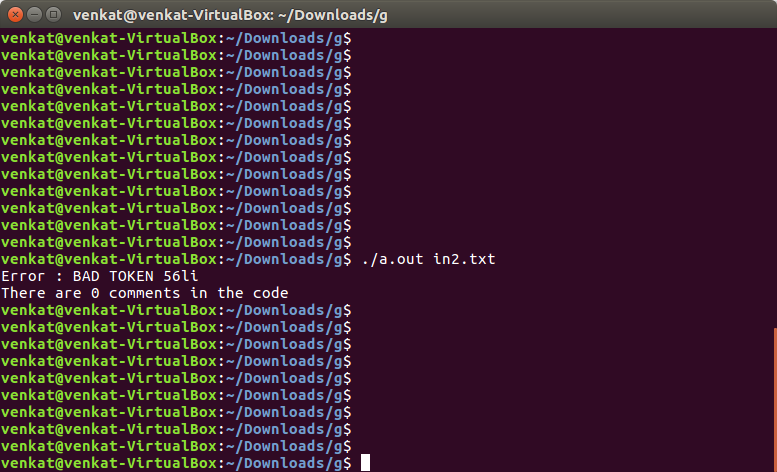
Output

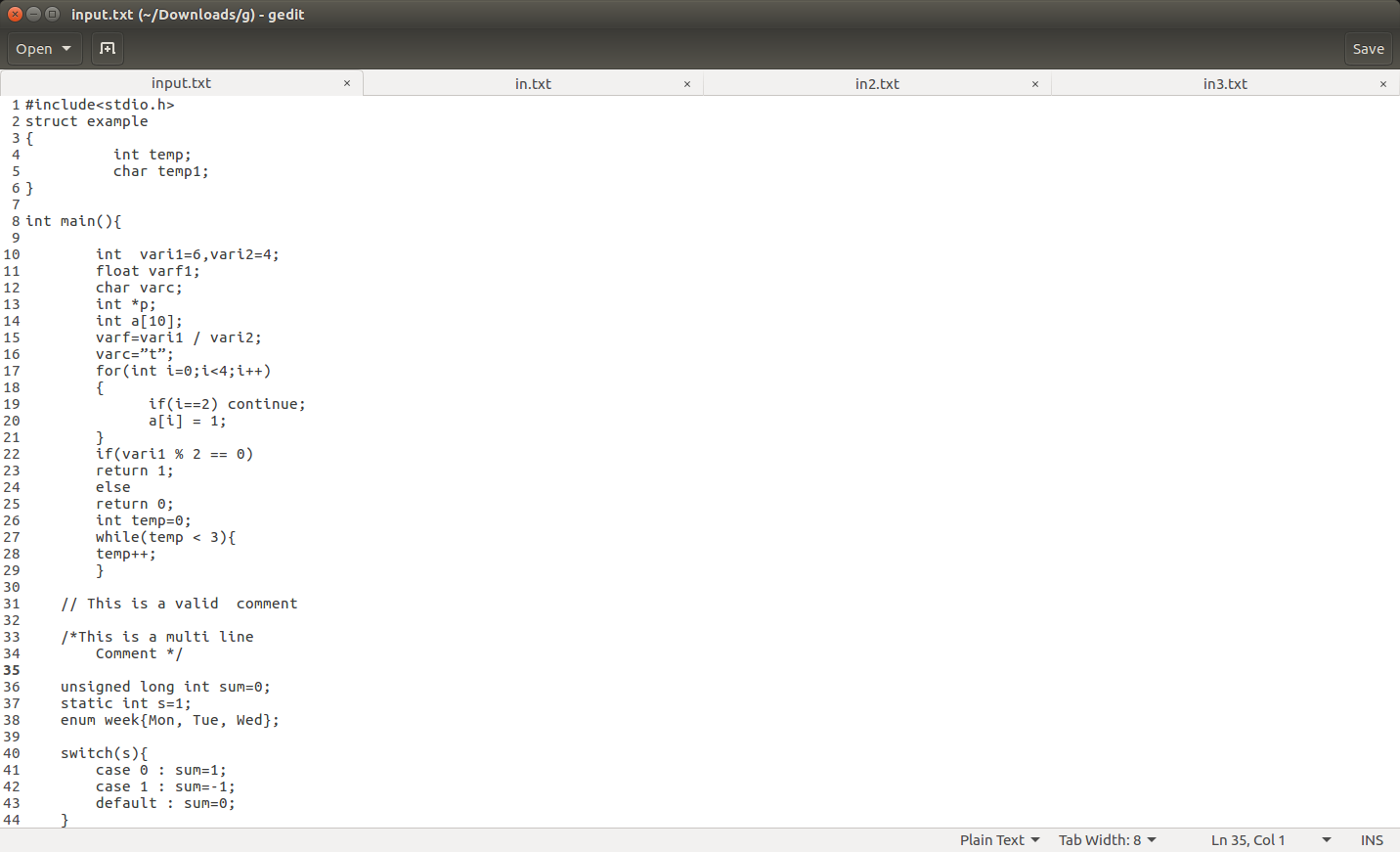


Test case 2



Output

Test case 3



Output

