

**COSC4461: Programming Languages**

**Fall 2024**

**Instructor: Marius Nagy**

**Course Project: Parser and Scanner for Complex Number Expressions**

**Group:**

**Mohammed Alqadda, 202002837**

**Kanaan Abukhadra, 202101747**

**Section 101**

**December 4, 2024**

**Project Objective:**

Implement a Scanner/Parser for a simple language for expressions involving complex numbers using FLEX (Fast Lexical Analyzer Generator) and Bison (parser generator). FLEX: A tool for generating scanners (lexical analyzers).

**Our Implementation:**

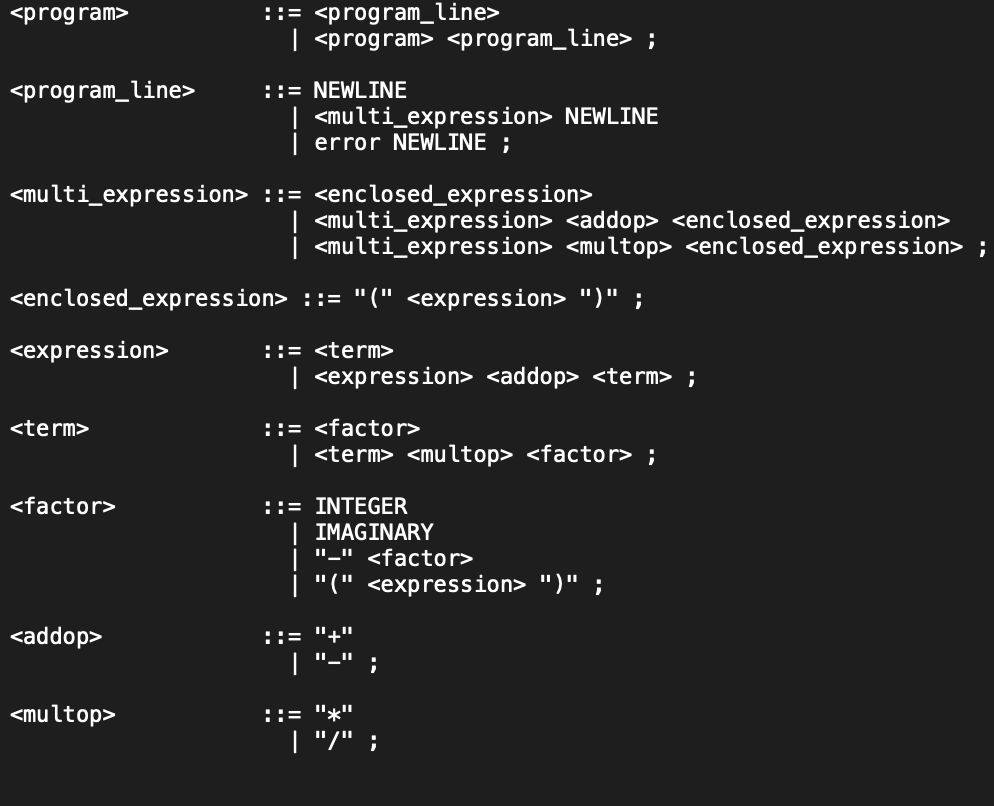
Our Language supports all four basic arithmetic operations +, -, \*, / between complex numbers in the form (a ± bi), where a and b are integers. Numbers that are missing either the real component (a) or the imaginary component (bi) are considered valid. Also, Parentheses for grouping.

All flex files are divided into three sections separated by %%. **Definitions Section**: Contains definitions and any C code needed (e.g., #include statements). **Rules Section**: Where the rules for scanning tokens are defined. **User Code Section**: Contains additional code (e.g., yywrap()).

A screenshot of a computer program

Description automatically generated Figure 1, is our flex file, highlighted in red is our definition section, where we added (#include "complex\_parser.tab.h"), this is for ensuring proper communication and type compatibility between the lexer and parser. In the rule section highlighted in green, we have our tokenizing rules, important example is the imaginary number, where we defined it as [0-9]+i to return IMAGINARY, which means any numbers from 0-9 of any length followed by i, should IMAGINARY. In the User code section highlighted in blue, we have yywrap() which is a Flex function called when the lexer reaches the end of input. Returning 1 tells Flex to stop scanning.

Figure 1 Our Flex File (Scanner), complex\_lexer.l

A screenshot of a computer program

Description automatically generated Figure 2, is our Bison file, highlighted in red is our definition section, where it contains definitions like parse.error verbose which gives a more detailed error parse errors, token declarations, like INTEGERS and IMAGINARY, and any C code needed, like yyerror() which handles syntax errors. In the rule section highlighted in green, we have our grammar rules for handling complex numbers, see figure 3.In the User code section highlighted in blue, we have yyerror() which is called when a parse error occurs. And the main(), which prints instructions on how to use the program, and calls yyparse() to start the process of the program.

Figure 2 Our Bison File (Parser), complex\_parser.y

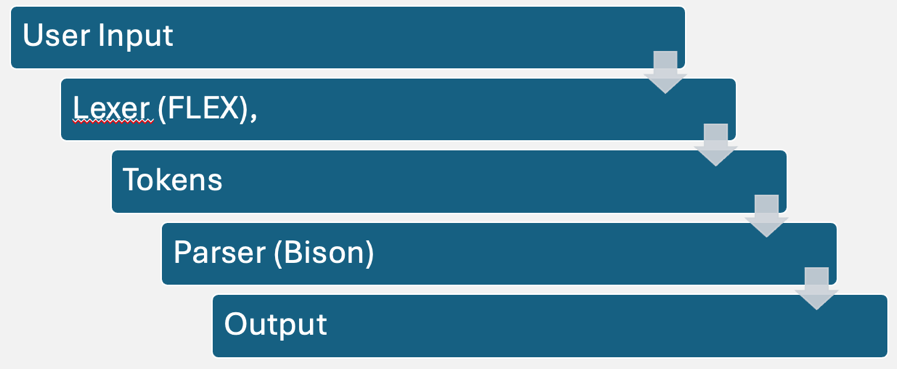
The process of the program and how it works, is shown in simple way in figure 4. Input: User enters a mathematical expression with complex numbers. Then, Lexer (Flex): scans the input and creates tokens like INTEGER, IMAGINARY, PLUS, and LPAREN and send it ****to the parser. After that, Parser (Bison): Does syntax analysis on the grammar rules. Validates expressions, ensures parentheses, and checks operator precedence. Lastly, the output in our case checks if the mathematical expression with complex numbers is valid or not.

Figure 3 Our Grammar in Backus-Naur Form (BNF)

Figure 4 flow from input to output

Parse Tree based on our grammar showing a valid Complex Number, (5+4i)\n. figure 5

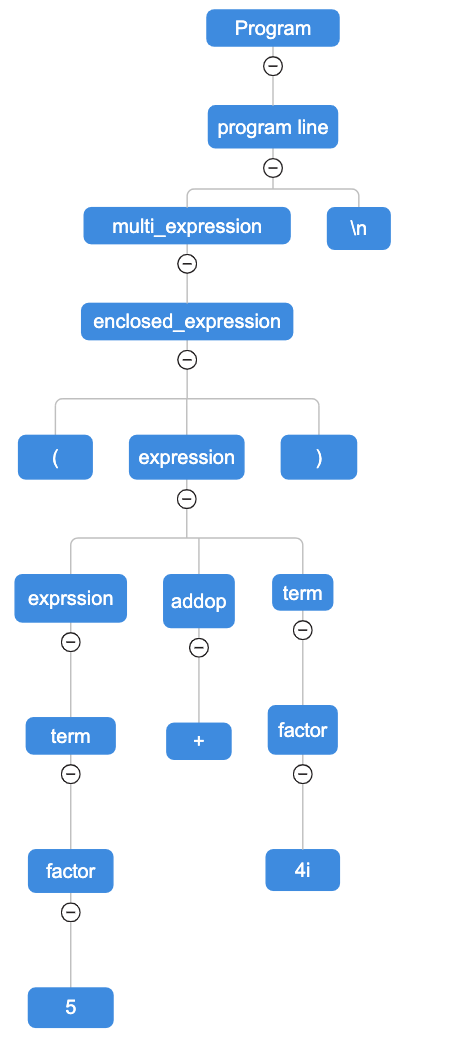


Figure 5 Parse Tree for (5+4i)\n

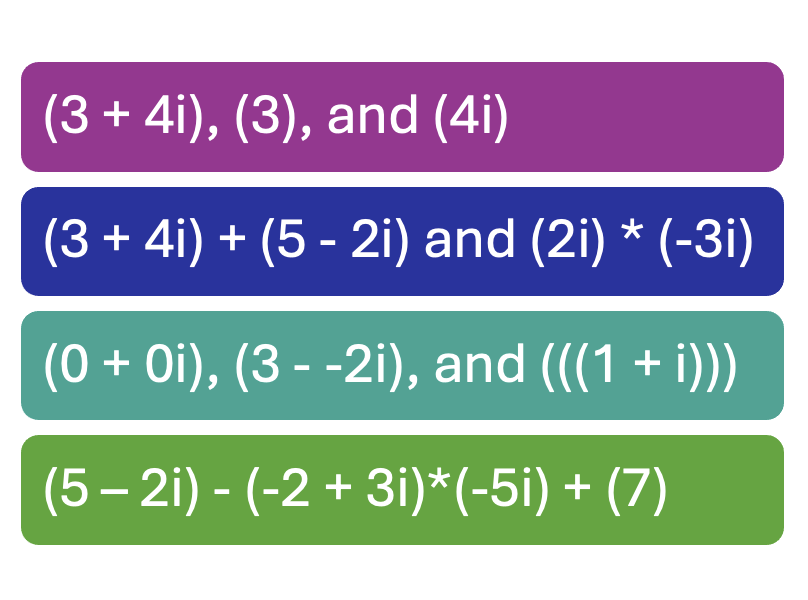
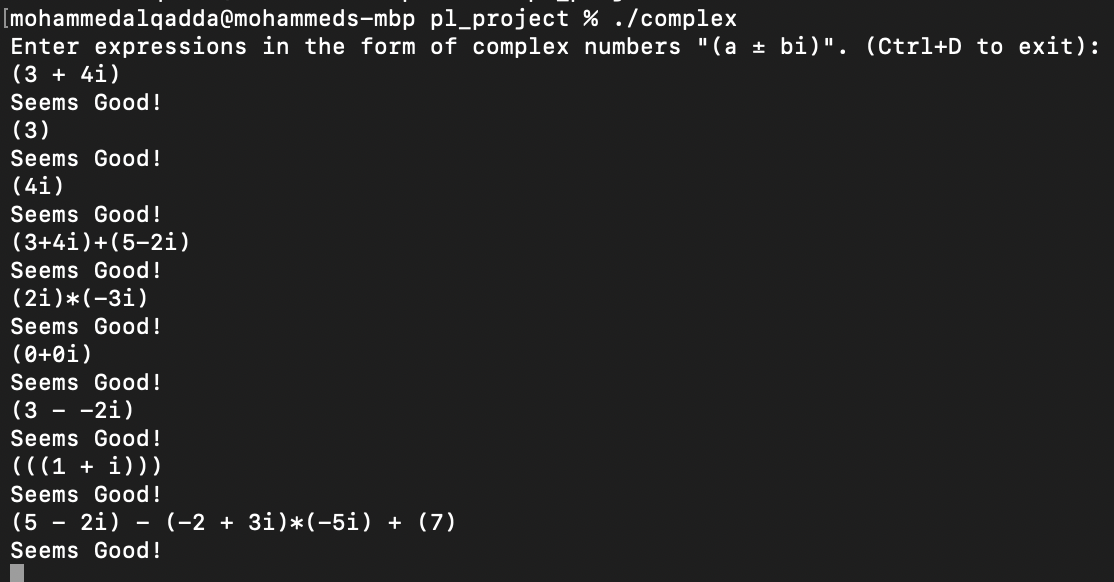
**Testing the Program**:

Figure 6 Outputs of Valid Examples

Figure 7 Examples of Valid Inputs

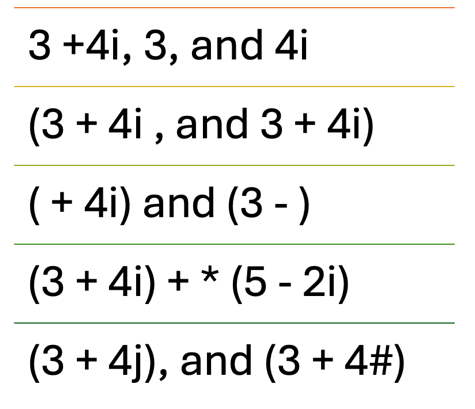
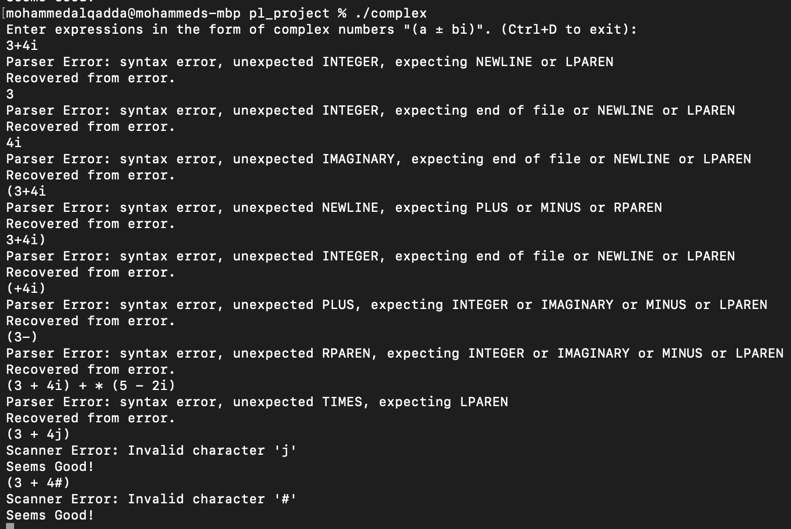


Figure 8 Outputs of Invalid Examples

Figure 9 Examples of Invalid inputs

**How to use Flex and Bison:**

You must first do the following,

* Installing FLEX and Bison:

For windows,

http://gnuwin32.sourceforge.net/packages/flex.htm

http://gnuwin32.sourceforge.net/packages/bison.htm

For macOS, (assuming you have brew)

brew install flex

brew install bison

* Creating needed files

FLEX files end with .l

Bison files end with .y

* After creating, run these commands,

bison -d filename.y

To generate the parser code. we get,

filename.tab.c and filename.tab.h

flex filename.l

To generate the scanner code and get,

lex.yy.c

**gcc lex.yy.c filename.tab.c -o programname**

To get an executable program. Which gives,

programname

The final executable, To run, ./programname

Link to our github project: <https://github.com/MFQ7/PL-Project/tree/main>