Name: Minh Tuan To
Student ID: 40114920

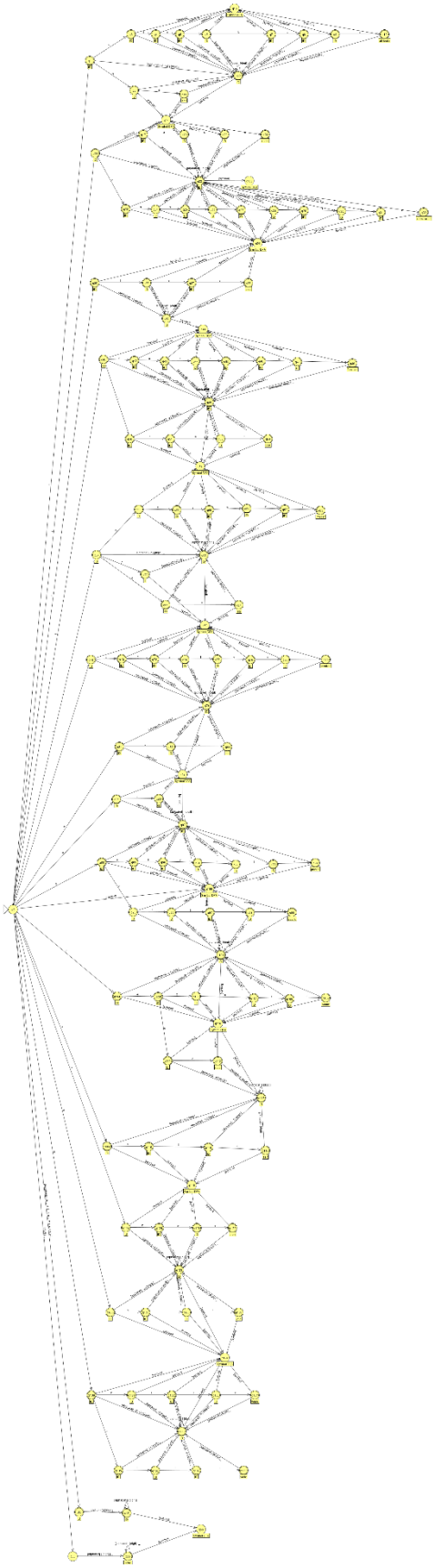
Lexical Analyzer report

Section 1: Lexical specifications

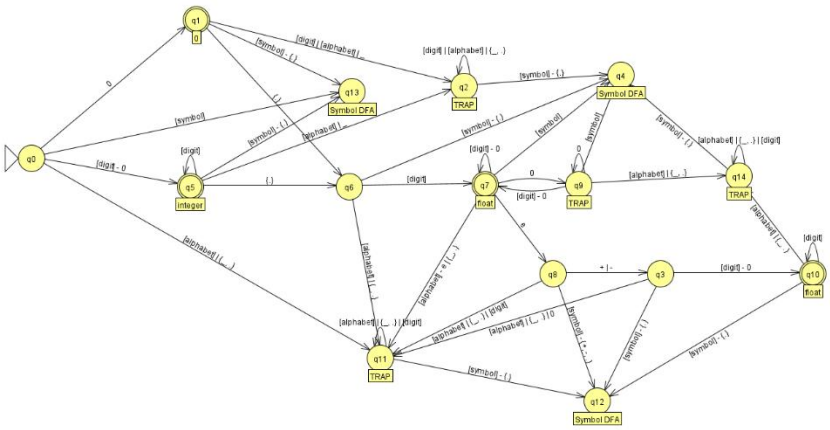
- Alphabet : From “a” to “z” and from “A” to “Z”
- Digits: From 0 to 9
- Symbols accepted: “=”, “<”, “>”, “+”, “-“, “*”, “/”, “(“, “)”, “{“, “}”, “[“, “]”, “,”, “.”, “;”, “:”
- Regular expressions:
    - Identifier: [a..z | A..Z] ([a..z | A..Z] + [0...9] + _)*
    - Integer: ([1..9] ([0..9])*) | 0
    - Float: (([1..9] ([0..9])*) | 0) . (0 | ([0..9])* [1..9]) (null | e (null | [+|-]) (([1..9] ([0..9])*) | 0))
      i.e., (*integer*) . (*fraction*) (null | e (null | [+|-]) (*integer*))
- Reserved words: or, and, not, integer, float, void, class, self, isa, while, if, then, else, read, write, return, localvar, constructor, attribute, function, public, private
- Comments:
    - Block comment enclosed by “/*” and “*/”
    - Inline comment starts with “//” and end with next-line character
- Changes to existing lexical specifications:
    - No nested block comments allowed.
    - Integers and floats starting with 0 are considered as errors.

Section 2: Finite state automaton

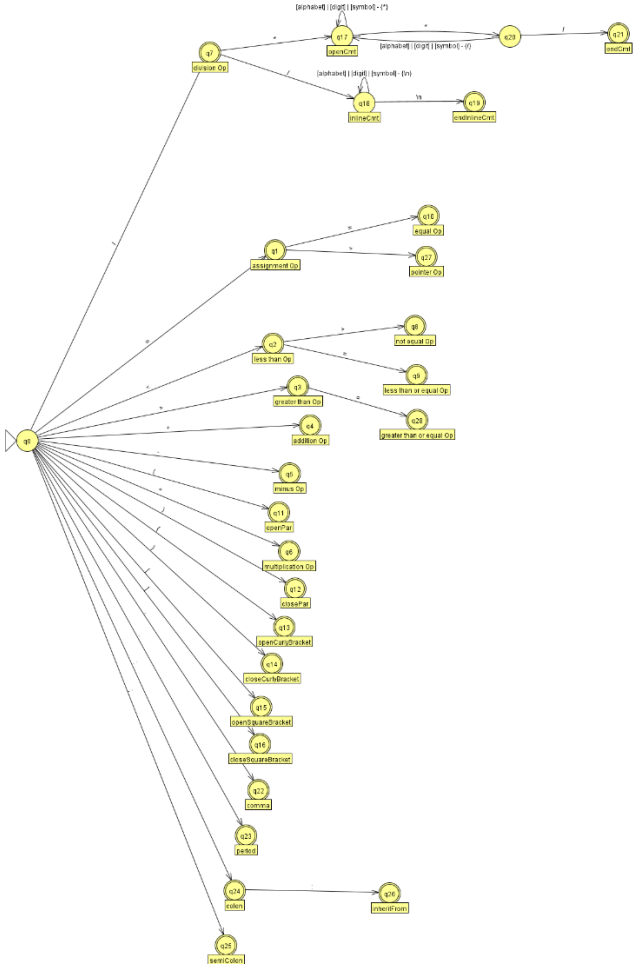
- DFA for identifier and reserved words processing:
  Link for clearer image on Google Drive (can zoom in): https://bit.ly/3wED3rS



- DFA for float and integer processing:
  Link for clearer image on Google Drive (can zoom in): https://bit.ly/3DoLDio



- DFA for symbol and comments processing:
  Link for clearer image on Google Drive (can zoom in): https://bit.ly/3DM9BV5



- DFA explanation:
    - **Alpha DFA**: As the name suggested, this DFA is used for processing identifiers and reserved words. Whenever it encounters a symbol, it automatically switches to Symbol DFA, which is the DFA that processes symbols and comments. This DFA cannot switch to Number DFA because numbers can be a part of identifiers.
    - **Number DFA**: This DFA is used for processing digits to identify integers and floats within the code. Since the symbol "." is also part of float, this DFA does not switch to Symbol DFA when encounter this symbol. However, for all other symbols, this DFA automatically switches to Symbol DFA. This DFA cannot switch to Alpha DFA because there cannot be a number containing letters (except letter "e" with float), and all numbers and identifiers must be separated by a symbol, space, or line to be considered valid.
    - **Symbol DFA**: This DFA is used for processing symbols and comments. Unlike the other two DFAs, this DFA will switch to Number DFA whenever it encounters a digit or Alpha DFA whenever it encounters a letter. This DFA can also switch between states internally whenever two symbols are written side-by-side. This transition is not demonstrated within the DFA picture because it will cause confusion, but we can imagine that whenever a final state is reached, there is a lambda transition back to the initial state. The transition table would better demonstrates these transitions.

Section 3: Design

- As the DFAs are designed, I also divide the "processors" into three main processors (Alpha, Number, and Symbol)
- **AlphaProcessor.java** is responsible for processing identifiers and reserved words. It uses the transition table to decide which state of the DFA it should go to, and if the current state is a final state.
- **NumberProcessor.java** is responsible for processing integers and floats. It also uses the transition table to decide which state of the DFA it should go to, and if the current state is a final state. I also created an enum called NumType to keep track of the current number type (whether it is an integer or a float).
- **SymbolProcessor.java** is responsible for processing symbols and comments. Like the other two, it also use transition table for movements between states. It has an enum called Sym that can be used to identify which symbol is currently being parsed. Any letters, symbols, and digits will be ignored (while still being kept track) when there is a block comment or inline comment.
- **Processor.java** is the parent class of all three processors, and it contains common necessary methods so that I can utilize the polymorphism property of Java in driver file.
- **Type.java** is an enum that indicates which input token it is processing. This helps with transitions between states in driver file and help processors identify the token.
- **State.java** is an enum that indicates which processor is currently being used. "Start" state is a state where no processor is currently being used, and driver needs to determine which processor should be used next. Other than that, "Alphabet" state corresponds with alpha processor, "Number" state corresponds with number processor, and "Symbol" state corresponds with symbol processor.
- **OutputWriter.java** is a static class that is an utility that can be used by all other classes to print to either error token file or output token file. This class keeps track of line number and buffered

writers. Comment writer is special because for multiple line block comments, we need to keep track of the first line number, not the last line number.

- **Driver.java** is the main file that brings everything together. It first read in everything, and then it splits the input string into tokens. Each token is matched with a regex pattern and processed based on the current state. If a space or next-line character is encountered, it automatically goes back to the start state in order to be ready for the next token.

Section 4: Tools used

- **DFA drawer**: JFLAP is the chosen application because there are too many characters in the alphabet so AtoCC cannot process all of these characters. Also, I have experience with JFLAP from COMP 335 course, so I choose this tool to draw my DFAs manually.
- **Programming language**: Java is chose because I am very familiar with this language, and I have used it throughout 3 years. It is a safe language for me to work with and debug.
- **Java libraries**:
    - BufferedReader: This file reader package is chosen because it is very fast in reading text files. Because I cannot process each token concurrently with the file reader, I figured that I would choose the fastest reader possible so that my code is a little bit more efficient.
    - BufferedWriter: I choose this writer just because of performance reasons.
    - Pattern: I use this package to match tokens with regex patterns, so that I don't have to use string compare. This also works faster than string compare as far as I know.
- **Transition table drawer**: Microsoft Excel is chosen because it is easy to create a table with excel, and I can easily save it as .csv file.
- **Debugging tool**: Java breakpoint on Visual Studio Code, because I am using VSCode to write this program, and it is significantly better than writing println() statements.