CS 316 Spring 2020
Observe **course policies** in undertaking this project.
All programs must be written in Oracle Standard Edition compliant Java or ISO/ANSI standard compliant C++.

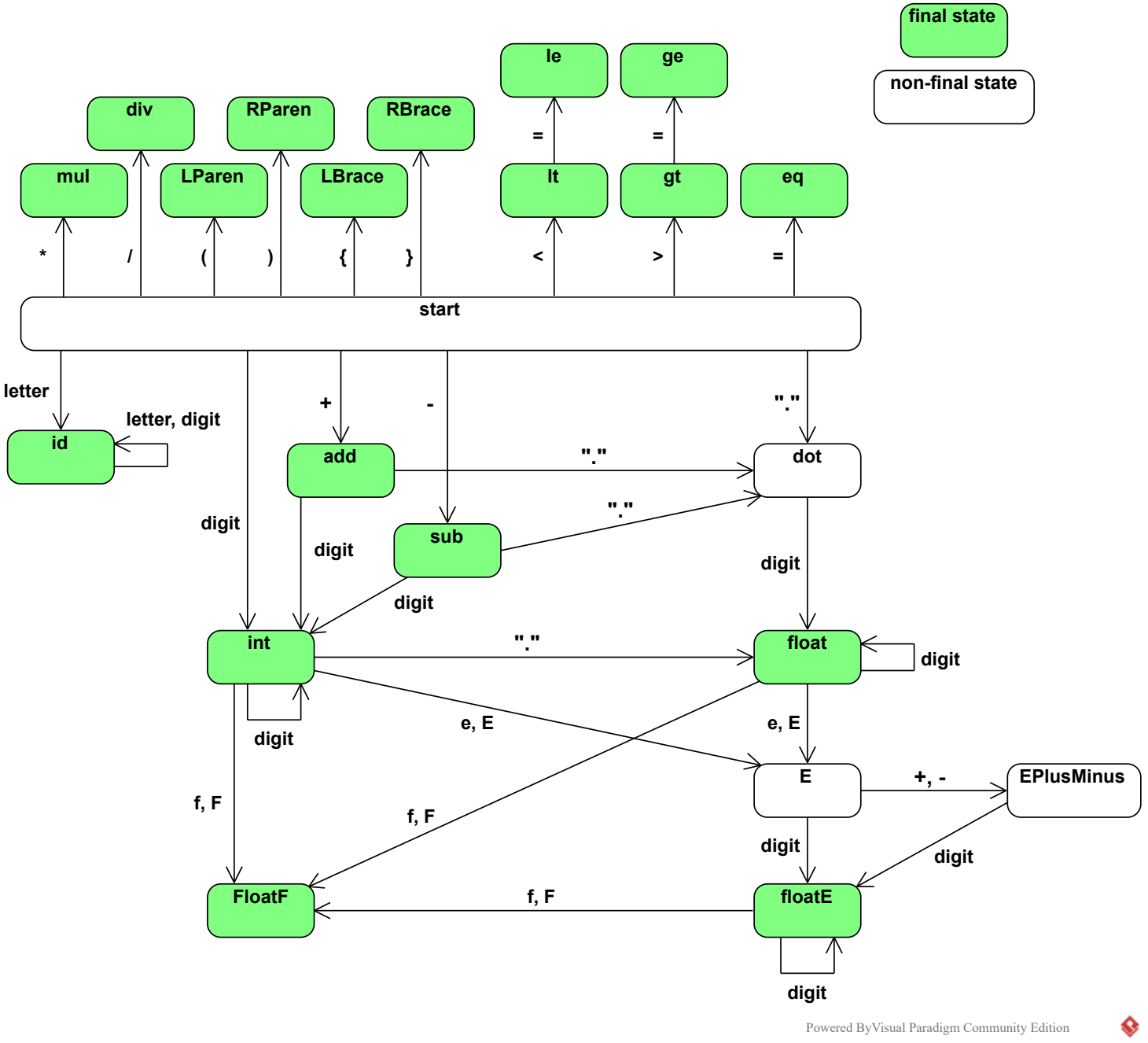
PROJECT 1: Lexical Analyzer
Due: 03/06/20, Friday, 11 PM
Late projects will not be accepted.

Consider the following EBNF defining 18 token categories ⟨id⟩ through ⟨RBrace⟩:

⟨letter⟩ → a | b | ... | z | A | B | ... | Z
⟨digit⟩ → 0 | 1 | ... | 9
⟨id⟩ → ⟨letter⟩ {⟨letter⟩ | ⟨digit⟩}
⟨int⟩ → [+|−] {⟨digit⟩}$^{+}$
⟨float⟩ → [+|−] ({⟨digit⟩}$^{+}$ "." {⟨digit⟩} | "." {⟨digit⟩}$^{+}$)
⟨floatE⟩ → (⟨int⟩ | ⟨float⟩) (e|E) [+|−] {⟨digit⟩}$^{+}$
⟨floatF⟩ → (⟨int⟩ | ⟨float⟩ | ⟨floatE⟩) ("f" | "F")
⟨add⟩ → +
⟨sub⟩ → −
⟨mul⟩ → *
⟨div⟩ → /
⟨lt⟩ → <
⟨le⟩ → "<="
⟨gt⟩ → >
⟨ge⟩ → ">="
⟨eq⟩ → =
⟨LParen⟩ → (
⟨RParen⟩ →)
⟨LBrace⟩ → {
⟨RBrace⟩ → }

⟨letter⟩ and ⟨digit⟩ are not token categories by themselves; rather, they are auxiliary categories to assist the definitions of the tokens ⟨id⟩, ⟨int⟩, ⟨float⟩, ⟨floatE⟩, ⟨floatF⟩.

According to the above definitions, the integers and floating-point numbers may be signed with "+" or "−". Moreover, the integer or fractional part, but not both, of a string in ⟨float⟩ may be empty. The following is a DFA to accept the 18 token categories.



The objective of this project is to implement a lexical analyzer that accepts the 18 token categories **plus the following keywords, all in lowercase letters only**:

if, then, else, or, and, not, pair, first, second, nil

These keywords cannot be used as identifiers, but can be parts of identifiers, like "iff" and "delse". In this and the next three projects, **the identifiers and keywords are case-sensitive**. The implementation should be based on the above DFA. Your lexical analyzer program should clearly separate the driver and the state-transition function so that the driver will remain invariant and only state-transition functions will change from DFA to DFA. The enumerated or integer type is suggested for representation of states.

The following keyword recognition method is adequate for this project.

1. Create 10 additional DFA states for the keywords.

2. The DFA initially accepts the keywords as identifiers.
3. Each time the DFA accepts an identifier, check if it is one of the keywords, and if so, move the DFA to the corresponding keyword state.

The lexical analyzer program is to read an input text file, extract the tokens in it, and write them out one by one on separate lines. Each token should be flagged with its category. The output should be sent to an output text file. Whenever invalid tokens are found, error messages should be printed, and the reading process should continue.

You may modify one of these sample Java programs into your solution; if you do so, modify the comments suitably as well.

Here's a sample set of test input/output files:

in1 | out1
in2 | out2
in3 | out3
in4 | out4
in5 | out5
in6 | out6
in7 | out7

Note that when the unexpected char is the newline char, the error message is displayed on the next line because my program, which is based on the sample lex analyzer, appends the newline char to the string read so far and displays it. You should make your own additional input files to test the program.

Since the purpose of this project is to reinforce, firsthand, the understanding of the internal mechanism of lexical analyzers built from finite automata as opposed to viewing them as black boxes, you are **not** allowed to use any library functions/tools for lexical analysis (like the Java StringTokenizer).

To make grading efficient and uniform, observe the following:

- The program must read the input/output file names as external arguments to the main function. How to set external arguments to Java main function in Eclipse.
- If Java is used, the main function to be invoked to run the program must be included in **LexAnalyzer.java** class, and if a package is used, the package that includes **LexAnalyzer.java** must be named **cs316project**.

The above token set is used for a small type-free functional language designed for our projects. Our project plan for the semester is to implement this language: a top-down parser in Project 2 and an interpreter in Project 3 and 4.

Submission

Your source program must be emailed to keitaro.yukawa@gmail.com with the subject header:

CS 316, Project 1, your full name

You may email the entire materials in a .zip or .rar compressed file.

The due date is 03/06/20, Friday, 11 PM. No late projects will be accepted. If you haven't been able to complete the project, you may send an incomplete program for partial credit. In this case, include a description of what is and is not working in your program along with what you believe to be the sources of the problems.