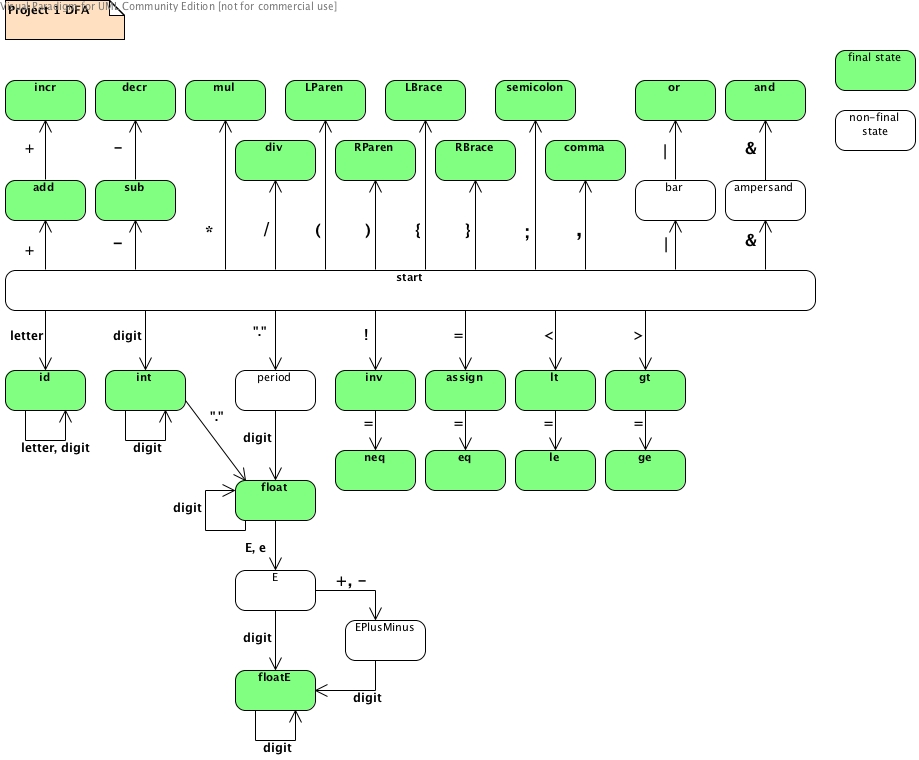
Consider the following EBNF defining 26 token categories <int> through <comma>:   
  
<letter>→ a | b | ... | z | A | B | ... | Z   
<digit>→ 0 | 1 | ... | 9   
<int>→ {<digit>}+  
<id>→<letter> { <letter> | <digit> }   
<float>→ {<digit>}+ "." {<digit>}  |  "." {<digit>}+  
<floatE>→<float> (E|e) [+|−] {<digit>}+  
<add>→ +   
<sub>→−  
<mul>→ \*   
<div>→ /   
<incr>→ "++"   
<decr>→ "−−"   
<or>→ "||"   
<and>→ "&&"   
<inv>→ !   
<lt>→ "<"   
<le>→ "<="   
<gt>→ ">"   
<ge>→ ">="   
<eq>→ "=="   
<neq>→ "!="   
<assign>→ =   
<LParen>→ (   
<RParen>→ )   
<LBrace>→ {   
<RBrace>→ }   
<semicolon>→ ;   
<comma>→ ,   
  
<letter> and <digit> are not token categories by themselves; rather, they are auxiliary categories to assist the definitions of the tokens <int>, <id>, <float>, <floatE>.   
  
According to the above definition of <float>, the integer or fractional part, but not both, of a string in <float> may be empty.   
  
The following is a DFA to accept the 26 token categories.



The objective of this project is to implement a lexical analyzer that accepts the 26 token categories **plus the following keywords, all in lowercase letters only**:

int, float, boolean, if, else, while, do, false, true

These keywords cannot be used as identifiers, but can be parts of identifiers, like "iff" and "delse". In this and the next three projects, we assume that 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 nine 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 state.

Alternatively, you may explicitly incorporate the step-by-step keyword recognition process into the state-transition function in the manner described in Problem 4 of [Exercise Set #2](http://picasso.cs.qc.cuny.edu/cs316/notes/Ex-2.html). (Many real-world lexical analyzers use this method for superior efficiency.) Extra credit will be given **provided** this method is implemented **correctly**. Use this method only if you feel able to handle the considerably increased number of states and state transitions, since a correctly working program using the above simpler method will get an A while a non-working program using this complex method will likely get a grade lower than A.   
  
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. The program should read the input/output file names interactively from the terminal or as external arguments to the main function; use a simple, standard file I/O method that works on any OS and in any program development environment.

**PS: you are not allowed to use any library functions/tools for lexical analysis (like the Java StringTokenizer).**