Key:
`*` = Kleene Star
`ϵ` = empty string
`l` = long
`f` = float

Andrea Gray
Dr. Siewert
CS 332
DUE: 8 FEB 2019

Assignment #2

1. Exercise 2.1: Write a regular expression to capture the following:

   a. Strings in C. These are delimited by double quotes and may not contain new line characters. They may contain double quote or backslash characters if and only if those characters are 'escaped' by a preceding backslash. The regular expression below shows that the string can be any combination of characters that DO NOT include newlines or non-escaped characters.

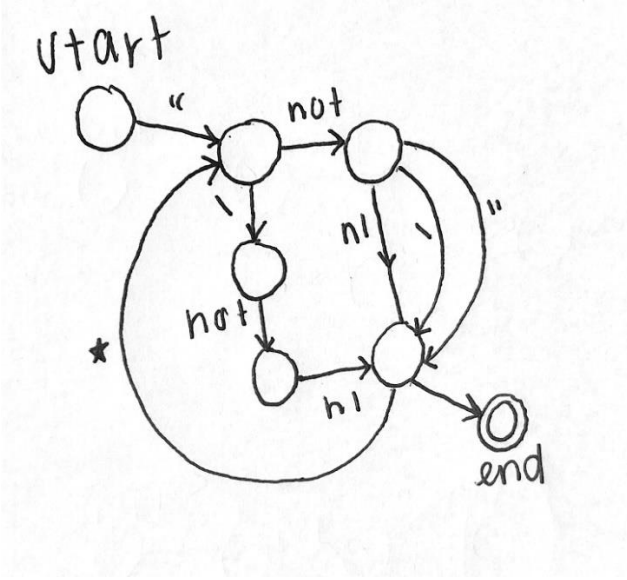
   ```
   String → "(not (newLine, ", \) | \ not(newLine)) *"
   ```

   c. Numeric constants in C. These are octal, decimal, or hexadecimal floating-point values.

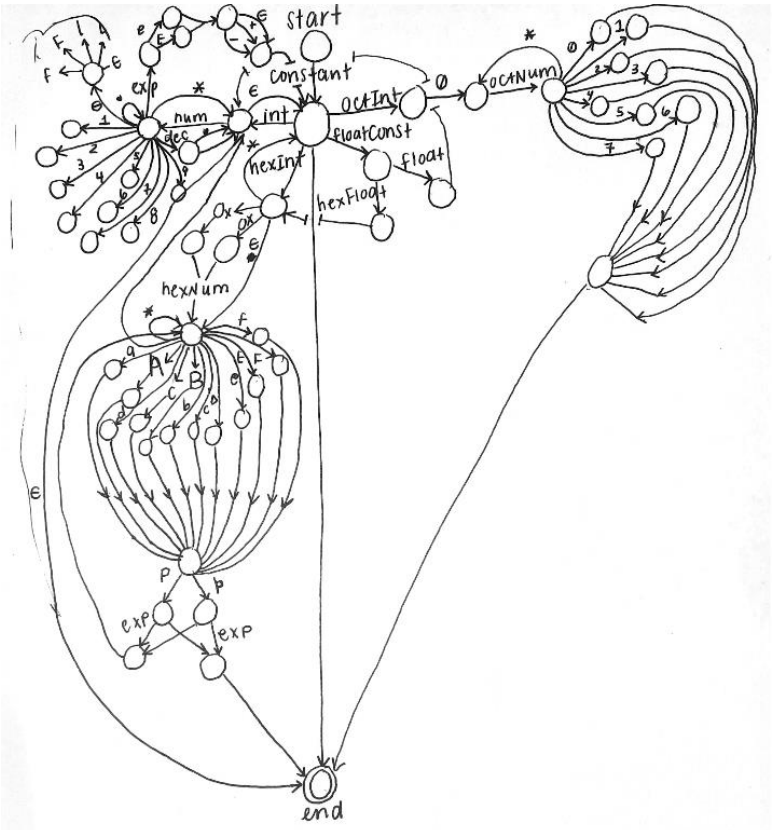
   ```
   constant → octInt | int | hexInt | floatConst
   int → num num*
   octInt → 0 octNum*
   hexInt → ( 0x | 0X) hexNum*
   dec → num* (. num | num .) num*
   exp → (e | E) (+ | - | ϵ) num
   num → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   octNum → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
   hexNum → num | a | b | c | d | e | f | A | B | C | D | E | F
   floatConst → (float | hexFloat) (f | F | l | L | ϵ)
   float → num* exp | num* dec (exp | ϵ)
   hexFloat → hexInt* (hexNum | ϵ hexNum) hexNum* (P | p) exp
   ```

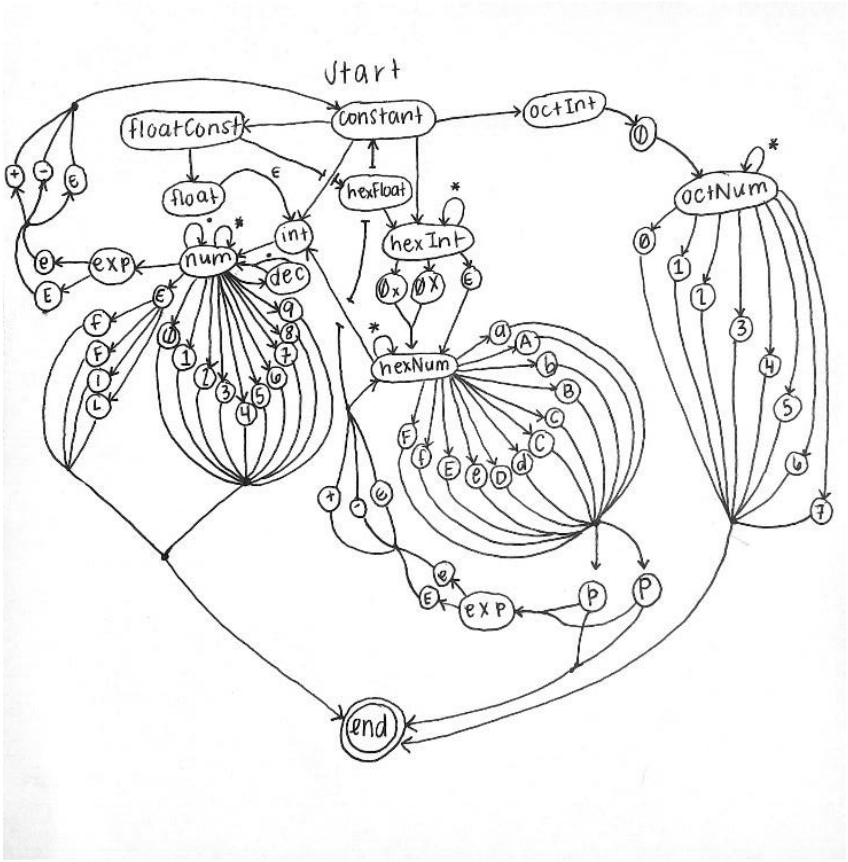
2. Exercise 2.2: Show the finite automata for the regular expressions above as a circles-and-arrows diagram.

   a. Finite automata for `string`.



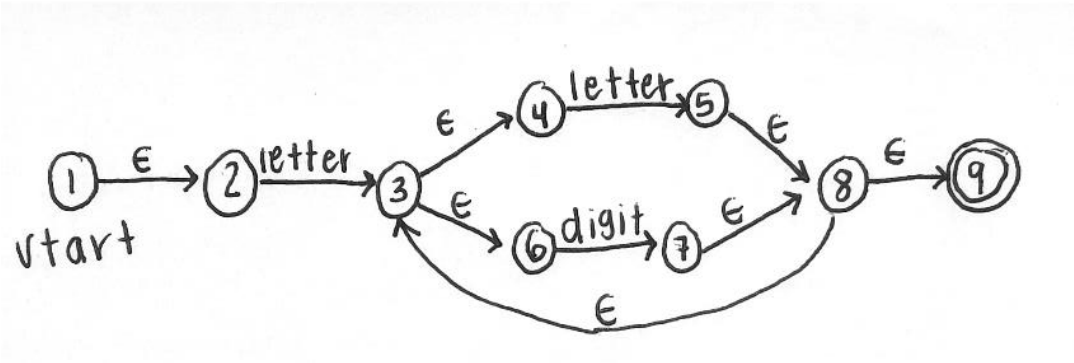
c. The finite automata below for `constant` is shown before and after “clean up”. I am not sure I cleaned up the image right, so I included both images.



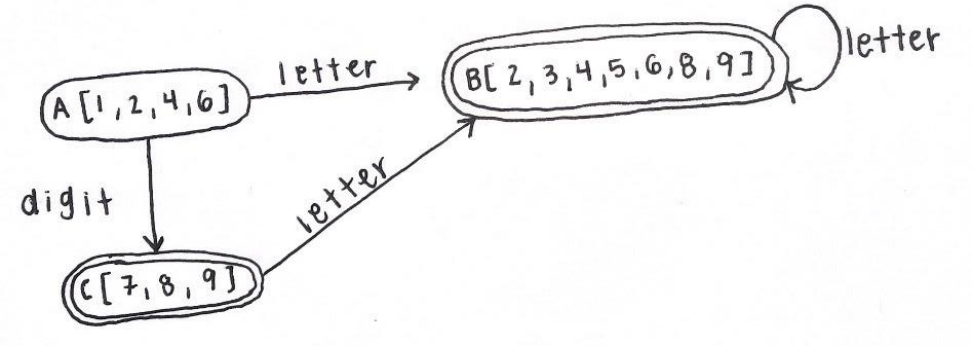


3. Exercise 2.4:

   a. The NFA that results from applying the construction of Figure 2.7 to the regular expression `letter (letter | digit) *` is shown below.



   b. The transformation illustrated by Example 2.14 to create an equivalent DFA is shown below.



4. A prototype for a calculator command line tool in C, as stated in Exercise 2.6, is attached in a separate .c file (Calculator.c). In comparison to the "bc" command line found on Linux and the calculator created for this assignment is still in its early prototyping phase. The calculator created for this assignment does not have as much exception handling as the "bc" command on Linux. While PEMDAS (or PMDAS for this assignment) is integrated, complex equations are not supported. Also, the division function is not perfected. I understand that in [1] a calculator function is used with switch…case statements. Upon further development of this program, the ability to match the functionality of the "bc" command on Linux in addition, subtraction, multiplication, and division is certain.

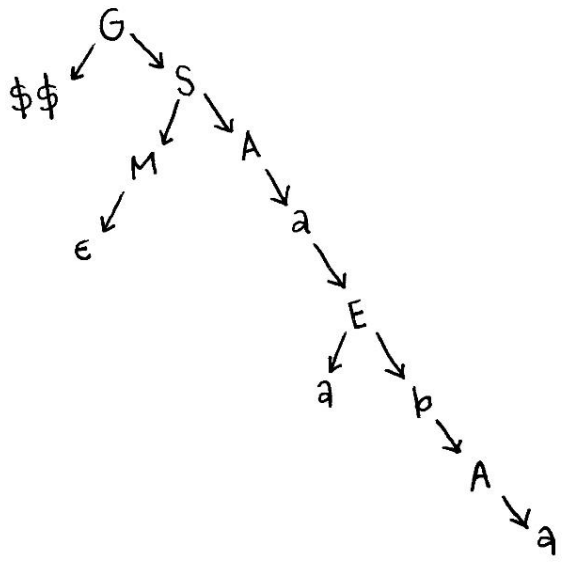
5. Exercise 2.12: Consider the following grammar

```
G → S $ $
S → A M
M → S | ϵ
A → a E | b A A
E → a B | b A | ϵ
B → b E | a B B
```

   a. In English, this grammar generates the string:
   `a a* b* (ab)* (ba)* (bb)* (bba*)* (a* (ab)*) (aa)* ϵ* $$`
   Which means that this grammar starts at G and then evaluates S. The $$ after S is "the end-marker pseudotoken produced by the scanner at the end of the input."[1] This means that the

user input ‘S’ which is then taken and evaluated to A and M. A and M are two concatenated items that are expanded respectively. A gives the option of going to E after the ‘a’ character is added to the output string. A also gives the recursive option to call itself twice after adding ‘b’ to the output string. E has three options which are going to B after ‘a’ is added to the output, going back to A after ‘b’ is added to the output, or ϵ which, as described in the key on page one, the empty string regular expression. B has two options which are go back to E after ‘b’ is added to the output string or recursively call itself twice after ‘a’ is added to the output. Finally, after the options for A have been completed, the grammar goes to M which simply gives the option for calling S again or outputting 0 + n empty strings. The Kleene stars in the generated string show where the grammar could loop and produce the same set of characters 0 + n times where n is any digit greater than 0.

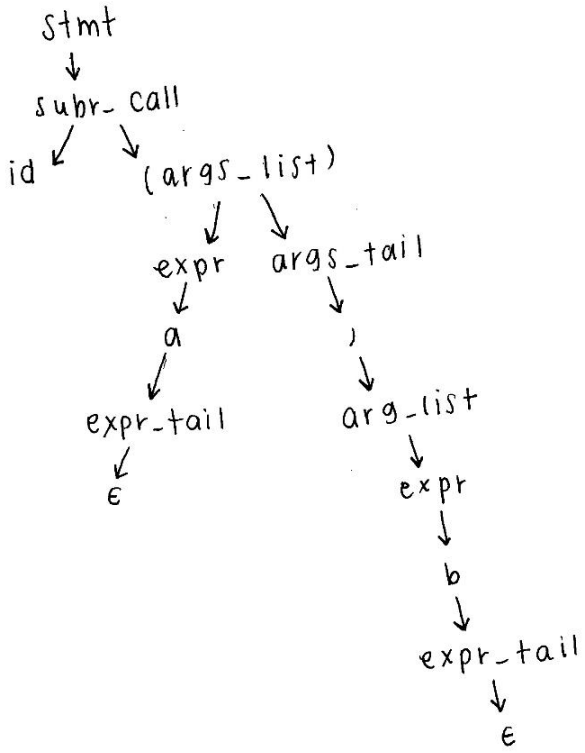
b. The parse tree for the string `a b a a` is shown below.



6. Exercise 2.13: Consider the following grammar:

```
stmt → assignment
     → subr_call
assignment → id := expr
subr_call → id ( arg_list )
expr → primary expr_tail
expr_tail → op expr
          → ϵ
primary → id
        → subr_call
        → ( expr )
op → + | - | * | /
arg_list → expr args_tail
args_tail →  ,  arg_list
          → ϵ
```

a. The parse tree for the input string `foo(a, b)` is shown below:



d. The grammar, given above, modified for LL(1) is:

```
program → stmt_list $$
stmt_list → stmt stmt_list | ϵ
stmt → id := expr | read arg_list | write expr
expr → term expr_tail | arg_tail
term → stmt | (expr)
expr_tail → op expr | ϵ
arg_tail → , expr | ϵ
operators → + | - | * | /
```

LL(1) is the left-left parsing with the compiler only looking 1 token ahead. Pages 50 and 72 in [1] were used to create the grammar.

7. For the final research project I have consulted with my partner, Thomas Rapp, and we have both agreed that an extensive research project on specific programming languages used and/or developed for quantum computing and the communication between user and hardware via software, as discussed with Dr. Siewert, would be an amazing research opportunity for not only the completion of the course, but for our own interests as well. research project early thinking. We plan to conduct research based off of the instruction Dr. Siewert gives us for the task. If he deems the subject inadequate though we will be creating our own domain specific interpreter. While this topic is interesting, the research process may prove to be less exciting than our ideal topic of quantum computing. To begin the research Thomas and I plan on delving into the structure and theories on which quantum computing is based. Once we are well versed in the basic hardware structure of a quantum computer, we will then research into how software will be implemented for these devices. Personally, I am very excited for this project and for the new information I learn on the subject.

8. Testing understanding of C++ name spaces, binding, and scoping rules by modifying and commenting on code given.

a. Modification of shifty.cpp so that it includes a right and left shift overloaded operator for hexadecimal and octal class numbers is submitted in a separate .cpp file (shifted.cpp). C++ does not provide this as a built-in float operator because the << and >> operators are part of a group of several versions of operators included in ostream. Furthermore, these operators, when overloaded, are only defined in the class, and not outside of it by ostream or any other standard in C++ (thank you Konrad Rudolph on stackOverflow for that explanation). The shifting operators are actually friend functions defined in ostream and are not provided by the language without the include statement for the operator. C++ knows to call the right operator because of not only the syntax of the operator's usage, but also because, as mentioned above, the operator is user-defined in the program they are developing. With this operator being locally defined C++ accesses the operator definition in the program before looking elsewhere. The "nm" dump symbols with the text "T" defined symbols identified is shown below. The "T" defined symbols are highlighted.

b. Examples from objex.cpp, including the template for swapping, is submitted in a separate .cpp file (piggy.cpp). In the piggy.cpp file the swap template demonstrates the swapping of piggy bank objects. The "nm" execution of piggy.cpp is shown below. The symbol name for the function instantiated to swap piggy banks based on nm output is unclear. I have included the "nm" output in an additional file, output.txt, to show the output of the function.

References:

[1] M. L. Scott, *Programming Language Pragmatics: Third Edition.* New York: Elsevier Inc, 2008.