COMP 370, Fall 2020
Program #3, Searching text for string that are in the language of a regular expression
Date Assigned: 10/1/2020
Date Due: 10/22/2020, 10PM
Possible Points: 100

For this assignment you will write a Python 3 program (in a file named `pa3.py`) that defines a class named `RegEx`. The constructor for this class takes as its parameter the name of a file, and it opens and reads an alphabet and a regular expression from the file. The constructor then converts the regular expression to an NFA, and then converts the NFA to a DFA. The class should also have a method called simulate that takes as its parameter a string, and determines if the string is in the language of the regular expression. Your program should behave exactly as described further below.

Your program must NOT use any modules/classes/libraries that do lexical analysis or parsing, or any of the other tasks that are part of the algorithms you are implementing. For example, don't use the regular expression (re) module in Python. The point of this program is to do the lexical analysis yourself, and develop a better understanding of how regex tools work. If you have any question about whether something you are doing is allowable, ask a question on campuswire (cateogry pa3). There is no credit for programs that violate this rule.

1 Initial Setup

Both you and your partner will need to clone the starter code for your group using git. The following assumes that you are using VSCode as your IDE. If you are using some other IDE, follow the instructions for that IDE to clone a repository, or just use command line git ("git clone url").

1. You will need your group number in what follows. Get this from the PSA Group Numbers file under the Programming Assignments tab on Blackboard. This file also has your assigned partner.

2. In VS Code, open the command palette and select the Git Clone option.

3. When prompted for the repository URL, enter the following, with X replaced by your group number (e.g. 7 or 12):
   ssh://git@code.sandiego.edu/comp370-fa20-pa3-groupX

4. Choose the Open Repository option in the window that pops up in the lower-right corner of the screen. The repository should contain the following files:

   - `pa3.py`: You will write your code in this file. It includes the header for the `RegEx` class that you will write, and the headers for the constructor and `simulate` methods that you must add.

- `test_pa3.py`: This program will test your `pa3.py` program. See the discussion below on this.
- `regexi.txt` for i = 1, ..., : Test regular expression files.
- `stri.txt` for i = 1, ... : Test input strings for the corresponding regular expressions.
- `correcti.txt` for i = 1, ... : Correct answer for the test input strings for the corresponding regular expressions.

5. Each programming team will have it own repository that has been initialized with the same starter code, so when you sync your code, you are sharing with your partners, but with no one else in the class. (I can see your repository also.)

6. Remember that if you close VSCode, then when you reopen it, you should see your repository. But if you dont see it, then just select `File->Open`, and then select the directory containing your repository.

7. I also recommend that you stage changes, commit those changes, and sync the changes periodically as you are working on the program, and certainly when you are done with a session with your programming partner. This ensures that you wont lose any of your work in case your computer gets lost or a file gets accidentally deleted.

2 Program Specifications

As said above, your program should be contained in a file named `pa3.py`. In this file you should define a class called `RegEx`. `RegEx` should have a constructor that takes as its parameter the name of a file, and initializes the `RegEx` from the contents of the file. (Specifications for the file format are further down.) The constructor should first convert the regular expression into an equivalent NFA, using the algorithm described in class and in the book. The constructor should then convert that NFA into an equivalent DFA, using the algorithm described in class and in the book. Your `RegEx` class should also have a simulate method that takes as its parameter a string, and determines if the string is in the language of the regular expression.

The project repository contains a `pa3.py` file that has the headers for the `RegEx` class, and for its constructor and simulate method.

Your `RegEx` constructor should:

1. Convert the regular expression to an equivalent NFA, using the algorithm described in class and in the textbook. (More on this further down.) Don't write the NFA to a file. Keep it in memory.

2. Convert the NFA into an equivalent DFA, using the algorithm described in class and in the textbook. Here, you can use the code you wrote for PA2, but do not write the DFA to a file – keep it in memory.

3. You should copy into `pa3.py` file your `NFA` class from `pa2.py`. You will need to modify the constructor of the `NFA` class, because you will not be reading it in from memory.

Your `RegEx` simulate method should:

1. Take as its parameter a string, and determine if the string is in the language of the regular expression. Since you have already converted the regular expression into an equivalent DFA, you can check if the string is in the language of this DFA by using code you wrote for PA1.

2. You should copy into `pa3.py` file your `DFA` class from `pa1.py`. You will need to modify the constructor of the `DFA` class, since you will not be getting the DFA from a file in this case.

You must implement the algorithm described above (and given in more detail below), and you must write all of the code yourself. To repeat what I said above: do not use any modules/classes/libraries that do lexical analysis or parsing, or any of the other tasks that are performed by the algorithms.

Here is more detail:

- The alphabet of the language appears by itself on the first line of the regular expression file. Every character in the line (not including the terminating newline character, or any space characters) is a symbol of the alphabet. The alphabet cannot include the letter e, the letter N, the symbol *, the symbol |, or the left or right parenthesis, as these have specific meanings in the regular expressions. See below for more on the format of regular expressions. (Note: in PA1 and PA2, the space character could be in the alphabet of the language, but here for this assignment we disallow it. This is so that spaces can be put into regular expressions, to improve readability, without being treated as symbols.)

- A regular expression appears by itself on the second line of the input file. See below for more on the format of regular expressions.

- If an invalid regular expression is encountered, your program should raise an InvalidExpression exception. This exception is already defined in `pa3.py`.

.

Here is detail on the format of the regular expressions that your program should handle. Recall the formal definition of a regular expression from section 1.3 in our text.

- The letter e represents ϵ (epsilon, the empty string) in a regular expression.

- The letter N represents $\emptyset$ (the empty set) in a regular expression.

- The character | represents $\cup$ (the union operator) in a regular expression.

- The character $*$ represents the star operator in a regular expression.

- The concatenation operator is implied in a regular expression. For example, in the regular expression $(ab^*)(c|ba)$ is short for $(a \circ b^*) \circ (c|b \circ a)$.
- Spaces can be embedded in a regular expression to help readability. So, for example, the above regular expression could be written $(a\ b^*)\ (c \mid ba)$.
- Recall that the star operator has highest precedence, then concatenation, and finally union. Parentheses are used to change the order of operation.

Here is how you will convert the regular expression into an equivalent NFA:

1. Parse the regular expression into an abstract syntax tree. In an abstract syntax tree, the interior nodes represent the operators, and the leaf nodes represent symbols in the alphabet. The children of the interior nodes are the operand(s) of the operator. Here is how you will construct the syntax tree (you may have done something like this in your COMP 120 or 151 class):
   (a) Create two initially empty stacks: a operand stack that will contain references to nodes in the syntax tree; and an operator stack that will contain operators (plus the left parenthesis).
   (b) Scan the regular expression character by character, ignoring space characters.
      i. If a symbol from the alphabet is encountered, then create a syntax tree node containing that symbol, and push it onto the operand stack.
      ii. If a left paren is encountered, then push it onto the operator stack.
      iii. If an operator (star, union, or implied concatenation) is encountered, then, as long as the stack is not empty, and the top of the stack is an operator whose precedence is greater than or equal to the precedence of the operator just scanned, pop the operator off the stack and create a syntax tree node from it (popping its operand(s) off the operand stack), and push the new syntax tree node back onto the operand stack. When either the stack is empty, or the top of the stack is not an operator with precedence greater than or equal to the precedence of the operator just scanned, push the operator just scanned onto the operator stack.
      iv. If a right parenthesis is encountered, then pop operators off the operator stack until the left parenthesis is popped off the operator stack. For each operator popped off the stack, create a new syntax tree node from it (popping its operand(s) off the operand stack), and push it onto the operand stack.
   (c) Empty the operator stack. For each operator popped off the stack, create a new syntax tree node from it (popping its operand(s) off the operand stack), and push it onto the operand stack.
   (d) Pop the root of the syntax tree off of the operand stack (and this should now be the only item on the operand stack).

(e) If any problems are encountered that indicate an invalid expression, then raise an `InvalidExpression` exception as described above.

2. Create an NFA from the abstract syntax tree by doing a depth-first traversal of the syntax tree. (Remember here that each node of the syntax tree is the root of a subtree that represents a regular expression.) For each node, an NFA is created that is equivalent to the regular expression represented by the subtree rooted at the node. If the node is a leaf node, then we have a base case, and the NFA is straightforward to create. If the node is an interior node (representing an operator), then the NFA is created from the NFA's of the child nodes using the constructions described in section 1.2 of the text (under "closure under the regular operations").

3. And now you have an NFA equivalent to the regular expression.

3 Working on the Program

You should work with your programming partner, using the pair programming software development technique.

As said above, the project repository contains test data for you. For each test regular expression there is the regular expression specification file (e.g. `regex1.txt`), a file containing test strings (e.g. `str1.txt`), one string per line, and a file containing the correct answers for each test string (e.g., `correct1.txt`), one answer (true or false) per line. (The lines of `correct1.txt` correspond to the lines of `str1.txt`.)

The project repository contains a Python program called `test_pa3.py` that tests your RegEx class on all of the test regular expressions in the files described above, and it reports the results. Your program must pass all tests before you submit your program for grading. If it fails any test, I'll return the program to you for further work. There is no partial credit concerning correctness of your program.

4 Program Rules

You should follow the following programming rules:

1. The algorithm implementations in your program should be clear, elegant, and efficient.

2. Your program should be well structured (see the guidelines below on functions/methods).

3. Commenting:

   (a) You should fill out the header at the top of the `pa3.py` file.

   (b) Each function/method should contain a docstring comment describing what it does. This explanation should be focused on its "input-output"

behavior; i.e., it should describe the conditions required to hold when it begins (preconditions) and the conditions that will hold after it finishes (postconditions). These conditions should be described in terms of the parameters (if there are any), and the return value, if there is one. You should describe any parameter whose meaning is not clear from its name, and describe the return value if its meaning is not clear from the name of the function/method. If a function/method implements a nontrivial algorithm, give a brief description of the algorithm in the function/method header (this could include a reference to a more complete description). Be sure that this "how it does it" part of the documentation is separate from the "what it does" part.

(c) When necessary to clarify their purpose, comment variables and constants.

(d) Keep the commenting of your code concise but descriptive. If your program is well designed and written, you will not have to comment every line. The function/method docstring comments, plus comments at the start of certain blocks of code, should be sufficient.

4. Code formatting:

   (a) Use whitespace to set off functions and methods within a class, and blocks of code within a function or method.

   (b) Be consistent with your formatting.

5. Functions/methods:

   (a) Each function/method should do only one task, but do it well. If your description of a function/method is rather long and complicated to describe, think about simplifying it by breaking its work into multiple functions/methods. Similarly, if the code for a function/method gets to be longer than a page or so (including documentation), think about breaking it up.

   (b) Keep the connections to functions/methods as simple as possible. Be wary of functions/methods with long parameter lists or control parameters that select what is to be done by the function.

6. Other:

   (a) Avoid using global variables, except for constants and shared variables.

   (b) Avoid using literal constants, except for 0 and 1. When you use a constant in your program, define a symbolic constant to be that value.

   (c) Don't nest control constructs (conditional and loop instructions) too deeply. It can be hard to follow the logic of code that is nested very deeply. Use your judgement here, but one way to deal with deep nesting is to break out some of the code into separate functions/methods.

(d) Choose identifier names that reflect the purpose of the object that they refer to (whether they are variables, constants, function/method names, parameter names, or class names).

Failure to follow these guidelines will result in points off.

5 Submitting Your Program

To submit your program for grading, be sure that you've pushed it the repository, and post a note on campuswire, category `PA3_submit`, giving just your group number. (Questions about pa3 should go to category PA3.)

6 Late Penalties

The turn-in time for your program will be the timestamp on the last commit (for a correct program) that you did in your repository, and not the time you post your submit note on campuswire. If you turn in your program after the deadline, the following late penalties apply:

- After Thursday, October 22, 10PM, but before Thursday, October 29, 10PM, 5 point penalty
- After Thursday, October 29, 10PM, but before Thursday, November 5, 10PM, 15 point penalty
- After Thursday, November 5, 10PM, but before Thursday, November 12, 10PM, 25 point penalty
- After Thursday, November 12, 10PM, no credit.