CS 164 : Programming Languages and Compilers

PA1: A DSL for Simple Regular Expressions

Professor: Koushik Sen, prepared by Alex Reinking *Due: 2/13/17*

1. Introduction

In this homework assignment, you will implement a DSL for expressing and matching strings against a subset of regular expressions. To do so, you will use the NFA evaluation algorithm presented in class by completing an implementation provided by the instructors. This assignment will *not* be done with a partner; you must turn in your own, individual work.

This assignment has two parts. First, you will write a *recursive descent parser* for a small regular expression syntax. Second, you will implement a matching engine for strings using Ken Thompson's algorithm following the slides for the regular expression to NFA procedure, and the NFA simulation algorithm which have been included in the `docs` directory. You do **not** need to compile the NFA to a DFA.

1.1. Setup.

You will write your programs in Java. The staff has supplied a skeleton for you to complete on GitHub. Using your Github student account, create a private repository for this assignment and add the staff account, `cs164staff`, as a collaborator. Run

```
git clone <url>
```

where `<url>` is the url for your repository to clone it to your computer. Then, cd into the directory, and run:

```
git remote add staff git@github.com:cs164staff/PA1.git.
```

Lastly, run `git pull staff` to download the initial code.

1.2. Software.

To develop locally, you will need the following software installed:

- JDK 1.6 (or newer). Available from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Maven 2.2.1 (or newer, 3.x.y recommended). Available from `https://maven.apache.org/download.cgi`
- JUnit 4. This dependency will be installed automatically by Maven, but the documentation is available here: `http://junit.org/junit4/`
- (optional) A Java IDE such as IntelliJ IDEA (free community edition; ultimate edition free for students) that can manage Maven projects.

If you are running Mac or Linux, we suggest installing these tools from your package manager, for example Homebrew on Mac, or Apt on Debian/Ubuntu. Your mileage may vary with Chocolatey on Windows or Apt on the Windows Subsystem for Linux (WSL). We recommend, if possible, using an IDE on Windows.

1.3. **Testing.** This assignment uses Maven to build and run the tests. To do so, simply:

(1) Open a terminal and `cd` into your git repository
(2) Run `mvn test`. You should be in the same directory as `pom.xml`

This will take care of building all the source files beneath `src/main/java` and executing all the tests beneath `src/test/java`.

Your code will be tested for correctness and basic performance across thirty test cases. Ten will cover the correctness of the regular expression parser specifically, and twenty more will test the correctness of the NFA evaluation. You have been supplied with ten of the test cases, so that you can check your progress. Your program will be evaluated against the remaining twenty to compute the final grade.

1.4. **Submission.** To submit, simply push to origin with `git push origin`. You must tag a commit with `pa1final` using the `git tag` command so we know which commit to grade. If you do not tag a commit, we will use the latest before the deadline, which could potentially be worse than a better, but slightly late, submission.

2. Problem Details

2.1. **Regular Expressions.** You will implement a simple, small subset of a typical regular expression engine. Your engine must support *concatenation*, *alternation*, *quantifiers*, *escape sequences*, and *nested expressions*. The grammar for such expressions is:

```
expr   :: term                # alternation
        | expr '|' term
term   :: factor              # concatenation
        | term factor
factor :: atom                # (optional) quantifiers
        | atom '+'
        | atom '*'
        | atom '?'
atom   :: '\n' | '\t' | '\|'  # escapes
        | '\(' | '\)' | '\*'
        | '\+' | '\?' | '\\'
        | '(' expr ')'        # nested expressions
        | any character other # single characters
          than: |()*+?\
```

Thus, alternation binds the least tightly, then concatenation, and quantification. The precedence of these operators can be altered with parentheses that behave as *non-capturing groups* would in an industrial regular expression engine. You do not need to handle advanced regular expression features, such as character classes, bounded quantification (other than `?`), capturing groups, or back-references.

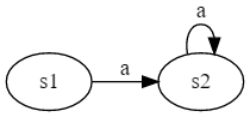
You must write your lexer / parser by hand. See `docs/SimpleParser.java` and `docs/SimpleParserWithParseTree.java` for examples.

2.2. **NFA Execution.** You will need to translate your parsed regular expressions into the NFA structures we've supplied. Read the documentation in `docs` provided with this assignment, and fill in the empty methods in `RegexParser` and `NFASimulator`. The files in the `docs` folder contain all the information necessary to translate regular expressions to NFAs and execute them.

2.2.1. *GraphViz.* It is helpful during development to visualize small NFAs generated by your program. You can produce a DOT representation of an `Automaton` by calling its `toString()` method:

```
digraph G {
    rankdir=LR;
    s1 -> s2 [label="a"];
    s2 -> s2 [label="a"];
}
```

(A) DOT code for `a+`



(B) Visualized output

You can generate the visualizations on the right from the descriptions on the left by either running GraphViz locally, or by using the following website: `http://dreampuf.github.io/GraphvizOnline/`.

3. Files and Directories

This is a standard Maven project, so you will find the following files:

- `pom.xml` - This is the Maven project file. There should be no need to modify it.
- `src/main/java/.../pa1` - All the relevant source files are located here. The only files you will modify are:
  - `RegexParser.java`: This class manages translating a `String` to an `Automaton`. Feel free to add any support classes or methods to this file, or to the `pa1` directory.
  - `NFASimulator.java`: This class takes an `Automaton` and a `String` and decides whether or not the NFA accepts the input. The method `matches` returns `true` if it does, and `false` otherwise.

  You might find it useful to add helper classes here.
- `src/main/test/.../pa1` - The files `NFASimulatorTest.java` and `RegexParserTest.java` contain the test cases you have been supplied. These files will be replaced by the full versions after submission, so feel free to add additional test cases here.

The project should compile as-is, but will always reject inputs, and will not pass any test cases.

4. Grading - 40 points

The rubric is as follows:

- 30 points. One point for each of the autograder tests, of which we supply ten.
- 5 points. General code cleanliness and style. Clear variable names, consistent spacing and brace conventions, no monolithic functions, etc.
- 5 points. README containing a description of any challenges you faced while completing this assignment.

5. Notes

Additional documentation about the classes is written in comments in the code, and the Javadocs are available in `docs/apidocs` but a few key points are stressed here.

- The `AutomatonState` class tracks its outgoing transitions. You can get the transitions for any given character by calling its `getTransitions` method. You can get the ϵ-transitions by calling its `getEpsilonTransitions` method.
- The `Automaton` class designates particular start and end states.
- Feel free to use Java's data structures and utility methods to speed development.
- However, no external libraries may be used (ie. do not modify `pom.xml`) and you must not use any code from the Internet.

A sample interaction for manually constructing an NFA for the regex `(a|b)+` follows:

```
// aIn -(a)-> aOut
AutomatonState aIn = new AutomatonState();
AutomatonState aOut = new AutomatonState();
aIn.addTransition('a', aOut);
// bIn -(b)-> bOut
AutomatonState bIn = new AutomatonState();
AutomatonState bOut = new AutomatonState();
bIn.addTransition('b', bOut);
// altIn -> { aIn, bIn }
AutomatonState altIn = new AutomatonState();
AutomatonState altOut = new AutomatonState();
altIn.addEpsilonTransition(aIn);
altIn.addEpsilonTransition(bIn);
// { aOut, bOut } -> altOut
aOut.addEpsilonTransition(altOut);
bOut.addEpsilonTransition(altOut);
// altOut -> altIn
altOut.addEpsilonTransition(altIn);
// final NFA
Automaton nfa = new Automaton(altIn, altOut);
```