COMP314
Assignment 1 - 2020

1. Purpose

The purpose of this assignment is to write a program which will allow a regular expression and a string to be inputted. The program should recognise whether the string is a member of the language which is denoted by the regular expression.

Regular expressions are stored as strings, and use similar notation to that of the lexical analyser generator application *JFlex*, with ordinary characters quoted and meta characters unquoted. Regular expressions can contain the following

Operands :

Single characters	:	enclosed in double quotes ("") e.g. "x"
Character classes	:	enclosed in square brackets ([ ]) and comprising only a range of characters e.g. [x-y]. Within a character class, the characters representing the lower and upper limits of the class are not quoted, and no whitespace is allowed.

Operators :

Alternation	:	| character (**not** the + character)
Concatenation	:	. character or whitespace
Kleene-closure (star)	:	* character
Positive-closure	:	+ character
Option	:	? character

Parentheses

Left and right parentheses can be used to override built-in operator precedence.

Whitespace

Whitespace is ignored within regular expressions.

An example is

("x" | "y")* . "x" "y" "y"

or [x-y]* . "x" "y" ."y"

Your code should perform the following :

(i) A regular expression is parsed and converted to an expression tree. There are four supplied classes, *RegLexer*, *Lexer*, *Token.class* and *RegExp2AST*, which provide the means to do this. You can carry out this step with code like the following :

```
String re = "...";   // the regular expression to be converted
//  Convert re to an expression tree RegExp (see 2.1)
RegExp r = (new RegExp2AST(re)).convert();
```

In the event of the regular expression containing a syntax error, the *convert()* method throws a *ParseException* exception. *ParseException* has methods *getMessage()* and *getErrorOffset()* which return respectively a message describing the error and the location in the expression at which it was detected.

The main method of the *RegExp2AST* class has some sample code to demonstrate the regular expression to expression tree conversion, and the detection and display of any errors.

(ii) The regular expression is then converted to an NFA using Thompson's construction. There are six supplied classes, *NfaState*, *Nfa*, *RegExp*, *RegSymbolExp*, *RegClassExp*, and *RegOpExp*, which provide the means to do this. You can carry out this step with code like the following

```
Nfa n = r.makeNfa();
```

(iii) Using the subset construction, the NFA built in (ii) must then be converted to an equivalent DFA. You must write the code for this.

(iv) Finally strings are tested to see if they are members of the language denoted by the regular expression using the DFA built in (iii). The method in which you input regular expressions and strings for testing must allow these to be read in from a text file. This can then be read in automatically every time you run your code, instead of having repeatedly to type in the data manually through the keyboard. Supplied is a sample text file, *TestData.txt*, which exercises a number of different regular expressions. The format of the file is as follows

Regular expression 1

Test string 1 for regexp 1

Test string 2 for regexp 1

....

Test string n for regexp 1

//

Regular expression 2

Test string 1 for regexp 2

Test string 2 for regexp 2

....

Test string n for regexp 2

//

....

2. Data Structures

2.1 Regular expressions

Regular expressions are compiled to expression trees with three types of nodes, *Symbol* nodes representing individual characters, *Class* nodes representing character classes, and *OpExp* nodes representing expressions. The base class is an abstract class *RegExp*, and the three node types are represented by the *RegSymbol*, *RegClass* and *RegOpExp* concrete classes, which are subclasses of *RegExp*. The instance variables in these classes show how the node types are represented.

You should not change any of the existing constructors, instance variables and constants in these classes, since the expression tree is built using them. *RegExp* has two abstract methods which is implemented in each of the concrete subclasses.

(i) The method ***makeNfa*()** constructs a NFA for that particular node type, using Thompson's construction, returning an object of type *Nfa* (see 2.3). E.g. the *makeNFA*() method in *RegSymbol* will use the Thompson's construction method for a single character to make an NFA for that character. This method is already implemented.

(ii) The *decompile*() method returns a string representing the original regular expression which was compiled to this expression tree node. E.g. the *decompile*() method in *RegClass* will return the string

[a-b]

where a is the lower bound and b the upper bound of the character class. This method is already implemented.

2.2 NFA states

Individuals states or nodes within an NFA are represented by objects from the *NfaState* class. The special properties of NFAs constructed with Thompson's construction allow for a simple, special-purpose design. Recall that such a construction results in an NFA with

(i) a single final state with no outgoing transitions.
(ii) a start state and internal states which have a single outgoing transition on a character, or the lower and upper bounds of a character class, or either one or two outgoing transitions on ε.

This allows us to use only the instance variables

```
char     symbol;  //  transition char (or lower bound of character
                  //  class)
char     symbol2; //  upper bound of character class (or
                  //  none)
NfaState next1,   //  only transition on char or 1st transition on ε
         next2;   //  2nd transition on ε (or null if none)
```

The drawback, of course, in using such a special-purpose structure is that it is only applicable to NFAs constructed by Thompson's construction, since other NFAs will not necessarily have the same properties as those produced by Thompson's construction. I've used such an implementation in the supplied *NfaState* class.

2.3 NFA

An individual NFA is represented by the *Nfa* class.

2.4 DFA

An individual DFA is represented by the *Dfa* class. The subset construction constructs a DFA in the form of a transition table, so this is the obvious form of structure to use. You can, however use your own design for this.

3 Comments

All the algorithms necessary for the subset construction, such as, e.g. the ϵ-closure algorithm, are in the course slides. Some of them make use of sets, so the set-based data structures in the *Java Collections Framework* should prove useful.

4. Submission

The supplied assignment code is in the file *Assignment1.zip* on *Moodle* in a form of an *Eclipse* project, which you can then import via the *Import/Existing Projects into Workspace* facility into *Eclipse.*

This is a group project, for which the class has been divided into 28 groups, A -Z, AA and BB. You can see what group you're in on Moodle on the *NAVIGATION/Participants* page. You can then find the other members of your group by filtering the display on your group. One member of each group should become group leader and be responsible for submitting the assignment.

The completed assignment is to be submitted as a single exported ***Eclipse .zip*** project file(**not** *.tar* or *.rar* or any other archive format you can think of!), using *Eclipse's Export/Archive File* facility. You must ensure that your project file can be imported into *Eclipse* and run without any changes needing to be made to it. The markers don't have the time to do this for you! The name of your project file **must** be *GroupName _Assignment_1.zip*, e.g. *F_Assignment_1.zip*. The main method through which the

assignment is run **must** be in a class called *RunAssignment*.

You will need to submit the properly documented source code of your classes, together with a record of the testing you undertook to show the correctness of your code in the form of the input text file mentioned in section 1, e.g. *TestData.txt*. In addition, there should be a brief statement stating for each group member what he or she was responsible for. These must all be in the *Eclipse* project file. Submit through *Moodle* by clicking on the *Assignment 1* topic on the front page and uploading your single zipped project file. You can resubmit as often as you like, but only the last submission is kept. The due date is 22 November at 23h59.

5 Practical Example of Use of Regular Expressions and Grammars in Translation

As an aside, you might be interested that in the code that parses your entered regular expressions and translates them to the expression tree data structure, I make use of regular expressions to specify the tokens that make up your regular expressions, and a context-free LL(1) grammar (something we'll meet a little bit later in the course) to specify the syntax of your regular expressions and to drive the parsing and translation to expression tree format.

I use a tool called *JFlex*, which operates as a transducer which takes definitions of the tokens in your regular expressions in the form of regular expressions together with some Java code fragments to specify what *JFlex* should do with a recognised token. It then produces automatically a Java class with methods that enable the next token in the input to be extracted. If you're interested, the token specification is in the file *RegLexer.flex* and the Java class that *JFlex* produces is in *RegLexer.java*.

The code that parses and translates your regular expressions is in the file *RegExp2AST.java*. The productions in the grammar used for the syntax of the regular expressions appear in the comments to the methods that use each production.