COMP314W2
Assignment 1 - 2021

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. They 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 precedences.

Whitespace:

Whitespace is ignored within regular expressions, except where it is not permitted in character classes.

An example is

("x" |"y") [x-z]* ."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* and *RegExpr2AST*, 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 expression tree 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:

```
RegExp r = .......              // as in step (i)
Nfa n = r.makeNfa();
```

The main method of the *RegExp2AST* class also has some sample code to demonstrate the expression tree to NFA conversion.

(iii) Using the subset construction, the NFA built in step (ii) must then be converted to a DFA, using the subset construction. 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 you built in step (iii). You must also write the code to do this. The method in which you input regular expressions and strings for testing must allow these to read in from a text file. This text file can then be read in automatically each time you run your code, instead of having repeatedly to type in the data manually from the keyboard. This makes it easier for you to test your code (and for me to mark it!). Supplied is a sample text file *TestData.txt*, which exercises a number of different regular expressions and strings. In this file each regular expression has a number of strings to test. 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
//
.....

Regular expression n
Test string 1 for regexp n
Test string 2 for regexp n
.....
Test string n for regexp n
//

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 classes which are implemented in each of the concrete subclasses.

(i) The method *makeNfa*() constructs an 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* uses Thompson's construction for a single character to make an NFA for that character. These methods are already implemented for you.

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

[a-b]

where a is the lower bound and b the upper bound of the character class that *RegClass* represents. These methods are already implemented for you, though you should not need actually to call them yourself.

2.2 NFA States

Individual states 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 with 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 character (or lower bound of
                        // character class)
char        symbol2;    // upper bound of character class (or empty)
NfaState    next1;      // only transition on char or 1st transition
                        // on ε
NfaState    next2;      // 2nd transition on ε (or empty 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. I've used such an implementation in the supplied *Nfa* and *NfaState* classes.

2.3 NFA

An individual NFA is represented by the *Nfa* class. Recall that Thompson's construction requires only the start and the single final (accept) state, so that an object from the *Nfa* class only then needs the two instance variables

```
NfaState start;         // start state
NfaState accept;        // final (accept) state
```

2.4 DFA

An individual DFA is represented as an object from the *Dfa* class. The subset construction constructs a DFA as a transition table, so this is the obvious form of structure to use. The *Dfa* class stub has a suggested transition table defined in it. However since this is code you write yourself, you can use your own design if you wish.

3. Comments

All the algorithms necessary, 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 *COMP314W2 Assignment1 2021.zip* on *Moodle* in the form of an *Eclipse* project, which you can then import into *Eclipse* via the *Import/Existing Projects into Workspace* facility in *Eclipse*.

This is a group project, for which the class has been divided into groups A-Z and AA-AI. You've already been given what groups you're in. One member of each group should become the group leader and be responsible for submitting the assignment.

The completed assignment is to be submitted as a single exported *Eclipse .zip* file (not a *.tar* or *.rar* or any other archive format you can think of!), using *Eclipse's Export/Archive File* facility. You must ensure that the project file can be imported into *Eclipse* using the *import/Existing Projects into Workspace* facility by the marker without any changes having to

be made to it. The markers don't have time to do this for you! The name of your project file must be *GroupName_Assignment1.zip*, e.g. *F_Assignment1.zip*. The main method through which the assignment is run must be in a class called *RunAssignment*.

You'll 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. This can be in the form of the input file you used to used to test your code, e.g. *TestData.txt*. In addition there should be a brief statement indicating 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. The due date is 2 November at 23h59.

5. Practical Example of Use of Regular Expressions and Grammar 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 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. It 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 a Java class with methods that enable the next token in the input to be extracted. If you're interested, the token specifications are in the file *RegLexer.flex*, the Java class that *JFlex* produces is in *RegLexer.java* and the tokens are defined in the *Token* class.

The code that parses and translates your regular expressions is in the file *RegExp2AST*. The productions in the grammar appear in the comments to the methods that process each production.