Harvey Mudd College
Computer Science 60
Fall 2010

Assignment 10
Fun with Regular Expressions
Due. 11:59 p.m., Wed., 24 November 2010
Pair-programming is optional on this assignment
(Please work with a different partner from your frequent partner, if you do)

Regular expressions are a meta-language for specifying other languages. The languages they can specify are exactly the ones acceptable by DFAs (Deterministic Finite-State Acceptors). As such, they are relevant to sequential logic circuits. However, they are also frequently found in software, as a way of specifying search patterns for text. We have already encountered the three operators (union, concatenation, and star) of regular expressions in our specification of grammars. Regular expressions specify the entire language without productions. For this assignment, the API is specified, but the code is entirely up to you, although I will provide design suggestions.

Problem Statement: Construct a library that will allow one to create arc-labeled directed graphs (ALDGs) and, with start and finish nodes specified, produce a regular expression specifying the set of all paths from start to finish.

Note that an ALDG is similar to other directed graphs we have studied, but has labels on arcs. In our case, the labels are regular expressions, as defined by the following inductive rules. Typically only single letters are considered symbols, but this is a special case of the present usage.

Rule	Example
Every **symbol** is a regular expression. In our case, a symbol can be an arbitrary ascii string. A symbol denotes a language of a single 1-symbol string.	**0**, **1, a, b**, and **c**, are each considered to be single symbols. We also allow **abc** as one symbol.
The **empty-set** symbol is a regular expression. It denotes the empty language.	∅ is the empty-set symbol, but we shall print it as **phi** in ascii.
The **empty-string** symbol is a regular expression. It denotes the language of exactly one string, the empty string.	ε is the empty-string symbol, but we shall print it as **epsilon** in ascii.
If R and S are two regular expressions, so is their **concatenation** RS, denoting the language of strings formed by concatenating a string in the language of R with one of the language of S.	**0 1** denotes the language consisting of the single string 0 1.
If R and S are two regular expressions, so is their union R+S, denoting the **union** of the languages of R and S respectively.	**0** + **1** denotes the language consisting of two strings, 0 and 1.
If R is a regular expression, then R* is also, denoting the language of **any number of strings** from the language of R concatenated.	**(0** + **1)*** denotes the language consisting of any number of 0's and 1's in any order.

More examples:

Regular Expression	Meaning
0*10*10*10	All strings over {0, 1} containing exactly three 1's
(0 + 111)*	All strings over {0, 1} in which 1's only occur as a sequence of three 1's.
(1* (1 + 01 + 001))*(ε + 0 + 00)	All strings over {0, 1} that do not contain three 0's in a row
((0 + 1)(0 + 1)(0 + 1))*	All strings over {0, 1} that have length a multiple of 3.
(0* (1 (01* 0)* 1)*)* 0*	All strings that are a multiple of 3 in binary (not obviously)

Note that many versions of regular expressions use a | or a ∪ instead of +. We use + here because it stands out more clearly. Also, don't confuse our regex package below with the standard regular expression libraries in Java.

Package adlg API: (You need to use this to conform to our test cases.)

Constructor: **ArcLabelledDirectedGraph()** creates a new ADLG

void **addNode**(String *nodeName*) adds a node named *nodeName* to an ADLG

void **link**(String *fromNodeName, String single,* String *toNodeName*) adds to the ADLG an arc with label *single from fromNodeName to toNodeName*.

Regex **getRegex**(String *fromNodeName,* String *toNodeName*) returns a regular expression representing the set of all paths from *fromNodeName* to *toNodeName*. In calling getRegex, we are allowed to make the assumption that there are no arcs into *fromNodeName* nor are there arcs outgoing from *toNodeName*.

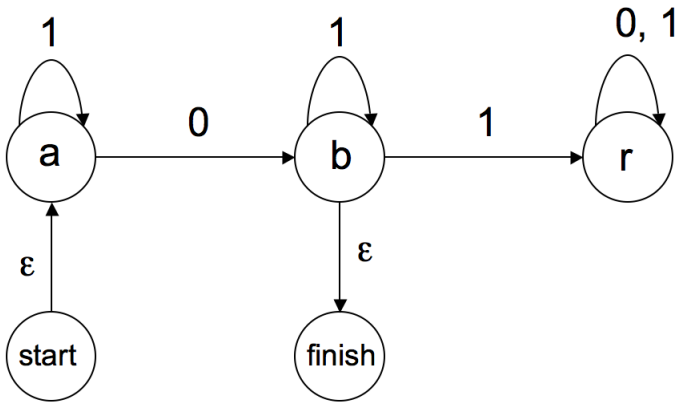
Suggested pacakge regex API:
Regex is the abstract base class. Other classes derived from Regex are:

- Concat
- EmptySequence
- EmptySet
- Single
- Star
- Union

Using this class structure makes it easy to build regular expressions syntactically. The toString() method for each of these determines the external look and feel. Create static functions in class Regex to build compound regular expressions. (Ask me if you don't understand this.)

Example Unit Test:

For this assignment, the API will be tested using jUnit tests. There is no main program. Below is an example of a test corresponding to the following graph:



The rationale for this configuration will be explained in class on Wed., Nov. 17, as will the method for deriving the regular expression, which is **1* 0 1*** in this case.

```
public void testGetRegex00()
  {
  ArcLabelledDirectedGraph graph = new ArcLabelledDirectedGraph();

  graph.addNode("a");
  graph.addNode("b");
  graph.addNode("start");
  graph.addNode("finish");

  graph.link("a", "1", "a");
  graph.link("a", "0", "b");
  graph.link("b", "1", "b");
  graph.link("b", "0", "r");
  graph.link("r", "0", "r");
  graph.link("r", "1", "r");

  graph.link("start", EPSILON, "a"); // EPSILON=new EmptySequence()
  graph.link("b",     EPSILON, "finish");

  System.out.println("\nArc Labelled Directed Graph:");
  System.out.println(graph);
  System.out.println("regex: " + graph.getRegex("start", "finish"));
  }
```

The output for this test is as follows, where the toString() of the regular expression is being printed:

```
Arc Labelled Directed Graph:
((a (1 a)(0 b))
(b (1 b)(0 r)(epsilon finish))
(r (0 r)(1 r))
(start (epsilon a))
(finish )
)
regex: 1* 01*
```