Representation and Operations on Numbers

Date of Submission: 20th March 11:55pm

1 Introduction

The purpose of this assignment is twofold:

1. Understand regular expressions as a way to *describe* strings with certain properties.
2. Implementing a *matcher* (more correctly a recognizer) for regular expressions. In the process, we shall have to program with trees.

2 egrep

We have seen some examples of regular in the class illustrated through a tool called `egrep`. `egrep` searches a file for a string described by a regular expression. The syntax of the egrep command is:

`egrep` *regexp filename*

It returns those lines in the filename which have a string that matches the regular expression. Here are the examples:

```
egrep '9\.' student-list.csv
```
Find all nine-pointers is the file student-list.csv

```
egrep '10\.' student-list.csv
```
Find all ten-pointers is the file student-list.csv

```
egrep '[4-6]\.' student-list.csv | wc -l
```
Count the number of students who are four, five or six-pointers.

```
egrep  '170050[0-9]{3}' student-list.csv
```
Find out lines containing some roll no.

```
egrep  '"[A-Z][a-z]*("|( [A-Z][a-z]*)*")' student-list.csv
```
Find out lines with a student name.

3 Regular expressions

In this assignment we shall implement a egrep like mechanism. To make life simple, we shall not consider the full range of regular expressions handled by egrep but a subset. Let us define this subset:

1. Any symbol is a regular expression matching only itself.
2. If r_1 and r_2 are regular expressions then $r_1|r_2$ is a regular expression. This matches any string that matches either r_1 or r_2.
3. If r_1 and r_2 are regular expressions then $r_1 r_2$ is a regular expressions. This matches any string that can be chopped up into two parts and the first part matches r_1 and the second matches r_2
4. If r is a regular expression, then $r*$ is a regular expression that matches those strings that can be expressed as a concatenation of 0 or more strings, each of which matches r.
5. If r is a regular expression, then $r+$ is a regular expression that matches thse strings that can be expressed as a concatenation of 1 or more strings, each of which matches r.
6. If r is a regular expression, then (r) is a regular expression that matches the same set of strings as r. Parenthesis are used for clarity.

The system that we shall design will answer the following question: *Given a regular expression, and a string, does the string match the regular expression?* I shall now give an overview of how we are going to answer the question. Consider the following regular expression—`((a|b)*bba)|cc*`. Does it match the string `babba`. To answer the question we shall construct a graph from the example (for now, never mind how) which looks like this:

Notice that the graph has nodes which are of three possible colours. Each such graph will have a unique green colored node and one or more red colored

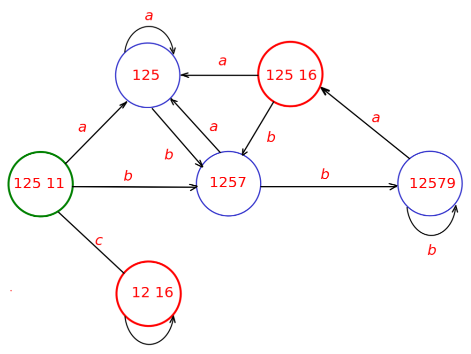

Figure 1: DFA for $(a|b)^*bba|c^+$

nodes. Now if we want to find whether a string such as `babba` matches the regular expression, we start with the green colored node and traverse the graph with the string. If the last symbol in the string takes us to a red node, the string matches the regular expression, else it does not. In fact, the graph has the interesting property that *the set of strings whose traversal ends in a red node is exactly the same as the set of strings that match the regular expression*. Your task in the project ends with the construction of the graph. The procedure to convert the regular expression to the graph is described in the accompanying document which will be discussed in the class and put up on moodle.

4 Structure of the assignment

We shall provide the following files to you.

1. The file `declarations.rkt` contains the `struct` declarations to represent (i) regular expressions in tree form (b) (ii) the graph and (iii) The edges (or transitions) of the graph.

2. The file `utilities.rkt` gives you several utility functions to make your life easier. For instance, it gives you a function `maketree` to convert regular expressions to tree form along with labels:

```
> (maketree "(a|b)*bba | cc*")
(Then (Or (Then (Then (Then (Star (Or (Literal "a" 1)
                                      (Literal "b" 2) 3) 4)
```

```
                  (Literal "b" 5) 6)
               (Literal "b" 7) 8)
            (Literal "a" 9) 10)
            (Then (Literal "c" 11)
            (Star (Literal "c" 12) 13) 14) 15)
          (Literal "#" 16) 17)
```

3. How is the graph represented? Well, here is the struct:

   ```
   (struct Graph(greennode nodes trans rednodes symbols)
     #:transparent)
   ```

   To see what it means, consider the function `buildGraph`. In fact, your assignment consists of writing this function. In my implementation, when I invoke this function as `(buildGraph (maketree "(a|b)*bba | cc*"))`, I get the result shown in the next page.

What you will do is to write the following functions:

1. You will construct the following functions: (i) `(buildNullable t)`, `(buildFirst t)`, `(buildLast t)` and `(buildFollow t)`. Using these functions, you will construct the function `(makeGraph regexp)`, which will convert a regular expression to the graph.

2. We shall also give you a function which will convert the graph to a visual form. But this later. First absorb what has been given to you.

3. Finally we shall also give you an additional function called `matches?` which will take a graph and a string and return the string "Regexp and string match" or "Regexp and string don't match".

5 A model implementation

This time I am going to provide you with a model implementation that is easy to use. Find a tar-zipped file called `model-implementation.tgz`. Its contents are

```
model-implementation --: test-model-implementation.rkt
                       : compiled--: regToDfa_rkt.zo
                                   : regToDfa_rkt.dep
```

```
(Graph
 (list 1 2 5 11)  ;The labels of the green node
 (list
  (list 1 2 5 11)(list 1 2 5)(list 12 16)(list 1 2 5 7) ; The nodes
  (list 1 2 5 7 9)(list 1 2 5 16))                      ; in the graph
 (list
  (Trans (list 1 2 5 16) "a" (list 1 2 5))       ; The edges
  (Trans (list 1 2 5 16) "b" (list 1 2 5 7))     ;
  (Trans (list 1 2 5 7 9) "a" (list 1 2 5 16))   ;
  (Trans (list 1 2 5 7 9) "b" (list 1 2 5 7 9))  ;
  (Trans (list 1 2 5 7) "a" (list 1 2 5))        ;
  (Trans (list 1 2 5 7) "b" (list 1 2 5 7 9))    ;
  (Trans (list 12 16) "c" (list 12 16))          ;
  (Trans (list 1 2 5) "a" (list 1 2 5))          ;
  (Trans (list 1 2 5) "b" (list 1 2 5 7))        ;
  (Trans (list 1 2 5 11) "a" (list 1 2 5))       ;
  (Trans (list 1 2 5 11) "c" (list 12 16))       ;
  (Trans (list 1 2 5 11) "b" (list 1 2 5 7)))    ;
 (list (list 12 16) (list 1 2 5 16))     ; Red nodes
 (list "b" "c" "a" "#"))                         ; Symbols in the
                                                 ; regular exp
```

Figure 2: Structure of the graph

Run drracket with `test-model-implementation.rkt`. This file has a list of functions that you could test for various inputs. Beware that this works for drracket 6.11. If your drracket version is different, let me know and I shall create a test package for it.