Technische Universität München **Winter term 2012/13**
I7
Prof. J. Esparza / J. Křetínský **23.11.2012**

Automata and Formal Languages – Programming Assignment

Due ~~18.12.2012~~ 7.1.2013

Your task is to write a program in Java/C++ that transforms regular expressions to automata and back and searches for occurrences of a given pattern. This amounts to implementing the algorithms from the lecture notes and extending them slightly. All referred files are accessible on the web of the course.

Grading

You shall work in pairs. An oral questioning will take place after evaluating the solutions to the second assignment (released after Christmas). Points will only be awarded after passing this questioning. Moreover, the source codes will be checked by the standard tools for plagiarism. For each part (this one and the one after Christmas), you can receive 3 points. For more details, consult the web page of the course. Points will be awarded for correct solutions to examples similar to the one below. Point awards are subject to correctness of the result and the time performance (the output has to be correct and produced within a few seconds).

Regular expressions

Let us define *extended regular expressions* using the following grammar:

$$r ::= r|r \mid r\&r \mid !r \mid (r) \mid (r)* \mid (r)+ \mid (r)? \mid (r)\hat{}n \mid r_1 r_2 \cdots r_k \mid a \mid . \mid \ : \mid @ \mid /$$

The semantics is given by the following rules:

$$\begin{aligned}
\mathcal{L}(r_1|r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1\&r_2) &= \mathcal{L}(r_1) \cap \mathcal{L}(r_2) \\
\mathcal{L}(!r) &= \Sigma^* \setminus \mathcal{L}(r) \\
\mathcal{L}((r)) &= \mathcal{L}(r) \\
\mathcal{L}((r)*) &= \bigcup_{i \geq 0} \mathcal{L}(r)^i \\
\mathcal{L}((r)+) &= \bigcup_{i \geq 1} \mathcal{L}(r)^i \\
\mathcal{L}((r)?) &= \mathcal{L}(r) \cup \{\epsilon\} \\
\mathcal{L}((r)\hat{}n) &= \mathcal{L}(r)^n \qquad n \text{ is and integer written in decimal} \\
\mathcal{L}(r_1 r_2 \cdots r_k) &= \mathcal{L}(r_1).\mathcal{L}(r_2).\cdots.\mathcal{L}(r_k) \qquad \text{concatenation of } k \in \mathbb{N} \text{ elements} \\
\mathcal{L}(a) &= \{a\} \qquad a \in \Gamma \\
\mathcal{L}(.) &= \Sigma \\
\mathcal{L}(:) &= \Sigma \setminus \Gamma \\
\mathcal{L}(@) &= \{\varepsilon\} \\
\mathcal{L}(/) &= \emptyset
\end{aligned}$$

where $\Gamma = \{A, B, \ldots, Z, a, b, \ldots, z\}$ (no numbers) and Σ contains Γ and all other symbols in files to be read.

The precedence of the operators is the following: | has the lowest priority, then &, then ! and the highest priority is assigned to $*, +, ?, \hat{}$. The parenthesis change the priorities as usual. For more details, consult the formal grammar in a file `regexp.g` on the web page. Example: $@|(a)?\&!a|(a)+$ is equivalent to $(@|\Big(((a)?)\&(!(a))\Big))|((a)+)$.

A *non-extended regular expression* is an extended regular expression given by

$$r := r|r \mid (r)* \mid r_1 r_2 \cdots r_k \mid a \mid . \mid @ \mid / \mid \ :$$

i.e. it does not use any intersection, positive iteration, maybe, power or complementation.

Functionality

All input and output files will contain regular expressions, one per each line. The last line may or may not be empty. Your jar file `re.jar` or executable `re` will be called with the following parameters and must have the following functionality:

- `-m` <file>
  For *each* extended regular expression in <file> in the order of appearance output into a newly created file <file>`.out` an equivalent *non-extended* regular expression. Aim for as small expressions as possible.[1] Size of the syntactic tree counts, i.e. the length of the expression not counting symbols "(", ")", "\".
- `-e` <file1> <file2>
  For *each* pair of extended regular expressions in <file1> and <file2> in the order of appearance print to the standard output "`y`" if they epress the same language and "`n`" if they do not.
- `-a` <file>
  For the *first* extended regular expression in <file> output into a newly created file <file>`.dot` an equivalent minimal determinisitic automaton in the format described below.
- `-s` <file1> <file2>
  For the *first* extended regular expression in <file1> print to the standard output the first matching expression in the text of <file2>, i.e. the one that ends at the earliest position in the text.
- `-n` <file1> <file2>
  For the *first* extended regular expression in <file1> print to the standard output the number of matching expressions in the text of <file2>. Note that there may be more matches ending at the same position.

The automata output format

Your program should create a description of a *minimal* deterministic finite automaton for the given expression ~~*without*~~ with or without a trap state. Since Σ is not explicitely given, you may also use "_" as a label of an edge meaning all elements of Σ that do not appear on other transitions from the same source (see the example below) and ":" as a label of an edge meaning all elements of $\Sigma \setminus \Gamma$. The output file has the following structure:

```
digraph G {
<initial state>
<list of final states>
<list of edges>
}
```

where:

- `<initial state>` is a line containing `<state>[shape=<diamond>];`
- `<list of final states>` is a sequence of lines each containing `<state>[peripheries=2];`
- `<list of edges>` is a sequence of lines each containing `<source> -> <target> [label=<label>];`

[1] Authors of the best solutions will receive a prize:-)

- `<source>`, `<target>`, `<state>` are decimal numbers
- `<label>` is either "_" or ":" or elements of Γ written in a sequence, no elements of $\Sigma \setminus \Gamma$ appear here.

You can check that your solution is displayed properly by running `dotty` on your output file.

An example

Consider the following files:

- `exps` contains

  ```
  a(.)*(a|b)
  @ | (a)? & !a | (a)+
  ```

- `text` contains

  ```
  a-a{}a
  ```

Upon calling `java -jar re.jar -m exps` a file `exps.out` is created with e.g.

```
a(.)*(a|b)
(a)*
```

The subsequent call with `-e exps exps.out` yields `yy` on the standard output. The call with `-a exps` produces a file `exps.dot` containing

```
digraph G {
0[shape=diamond];
2[peripheries=2];
0 -> 1 [label="a"];
1 -> 1 [label="_"];
1 -> 2 [label="ab"];
2 -> 2 [label="ab"];
2 -> 1 [label="_"];
}
```

The call with `-s exps text` outputs `a-a` and with `-n exps text` outputs ~~2~~ 3.

What to hand in?

By ~~**December 18**~~ **January 7** you have to hand in the following files into an svn that will be assigned to you shorlty. The topmost folder shall contain

- a compiled executable file `re` or executable jar file `re.jar`;
- folder `src` with source codes; the code should be well structured, easily readable and properly commented and documented, in particular every class and method should be (apart from comments in the code) immediately preceded by a comment on what it does;
- a file `description.txt` very briefly describing the structure of your source files, i.e. which file implements what;
- a file `howtocompile.txt` containing the command used to compile your files in order to get the executable; it may also contain any other comments from your side if necessary.

Note that the sources must be compilable on `lxhalle.in.tum.de` using the command written in `howtocompile.txt`. Otherwise, *no points will be awarded.*