Finite Automata from Regular Expressions — Yamada-McNaughton-Glushkov Construction

1 Aim of the exercise

The aim of the exercise is to reinforce knowledge of programs `flex` and `bison`, to reinforce skills of parsing, and broadening the knowledge of finite state automata.

2 Environment

Synopsis of `dot`:

```
dot -Tps < source > result.ps
```

The program under development can best be tested using the following pipe (put your expression in `echo`):

```
echo '0(0|1)*0' | ./z7c | dot -Tps > result.ps; gv result.ps &
```

3 Yamada-McNaughton-Glushkov construction

A nondeterministic automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of *states*, Σ is a finite set of symbols called the *alphabet*, $\delta : Q \times \Sigma \to 2^Q$ is the *transition function*, $q_0 \in Q$ is the *initial state* (*start state*), and $F \subseteq Q$ is the set of *final states* (*accepting states*). It is constructed from parts in a similar way to that in which an arithmetic expression is calculated. A characteristic feature of Glushkov automata that result from Glushkov construction is that that all incoming transitions for a state have the same label, all symbols present in a regular expression are numbered. For example, in an expression "010" the first zero is different from the second one. The whole expression is treated as $0_1 1_2 0_3$. A feature *Null*, as well as sets *First*, *Last*, and *Follow* are used. Feature *Null* is true if an empty sequence of symbols is recognized by the expression, i.e. for a regular expression $r \in RE$:

$$Null(r) \Leftrightarrow (\varepsilon \in \mathcal{L}(r)) \tag{1}$$

Set *First* is a set of numbered symbols that can appear at the first position in words belonging to the language of the regular expression, i.e. for $r \in RE$:

$$First(r) = \{\sigma \in \Sigma : (\exists w \in \Sigma^*)\sigma w \in \mathcal{L}(r)\} \tag{2}$$

Set *Last* is a set of numbered symbols that can appear at the end position in words belonging to the language of the regular expression, i.e. for $r \in RE$:

$$Last(r) = \{\sigma \in \Sigma : (\exists w \in \Sigma^*)w\sigma \in \mathcal{L}(r)\} \tag{3}$$

Set *Follow* is a set of pairs of numbered symbols that can appear in that order one immediately after another in words belonging to the language of the regular expression, i.e. for $r \in RE$:

$$Follow(r) = \{(\sigma_1, \sigma_2) \in \Sigma^2 : (\exists u, w \in \Sigma^*)u\sigma_1\sigma_2 w \in \mathcal{L}(r)\} \tag{4}$$

Values of *Null*, *First*, *Last*, and *Follow* for basic regular expressions $\emptyset$, ε, and $\sigma \in \Sigma$ are given in table 1. Having two regular expressions $r_1 \in RE$ and $r_2 \in RE$ as well as values of *Null*, *First*, *Last*, and *Follow* calculated for them, one can calculate those values for complex regular expressions like alternative $r_1|r_2$, concatenation $r_1 r_2$, and transitive closure r_1^* using the table 1.

Once one has values of *Null*, *First*, *Last*, and *Follow* for the whole regular expression, a finite-state automaton can be built. Each numbered symbol is associated with a different state, therefore **for each numbered symbol** in a regular expression, a **state** is created, and the symbol is stored in that state. The state number is then used instead of a numbered symbol.

As symbols are associated with target states of transitions, one has to create an additional state that will serve as the **initial state**. That state will be final if the value of *Null* for whole expression is true.

RE	*Null*	*First*	*Last*	*Follow*
$\emptyset$	*false*	$\emptyset$	$\emptyset$	$\emptyset$
ε	*true*	$\emptyset$	$\emptyset$	$\emptyset$
$\sigma \in \Sigma$	*false*	$\{\sigma\}$	$\{\sigma\}$	$\emptyset$
$r_1 \vert r_2$	$\mathit{Null}(r_1) \vee \mathit{Null}(r_2)$	$\mathit{First}(r_1) \cup \mathit{First}(r_2)$	$\mathit{Last}(r_1) \cup \mathit{Last}(r_2)$	$\mathit{Follow}(r_1) \cup \mathit{Follow}(r_2)$
$r_1 r_2$	$\mathit{Null}(r_1) \wedge \mathit{Null}(r_2)$	$\mathit{First}(r_1)$ *if* $\neg\mathit{Null}(r_1)$ *else* $\mathit{First}(r_1) \cup \mathit{First}(r_2)$	$\mathit{Last}(r_2)$ *if* $\neg\mathit{Null}(r_2)$ *else* $\mathit{Last}(r_1) \cup \mathit{Last}(r_2)$	$\mathit{Follow}(r_1) \cup \mathit{Follow}(r_2) \cup (\mathit{Last}(r_1) \times \mathit{First}(r_2))$
r_1^*	*true*	$\mathit{First}(r_1)$	$\mathit{Last}(r_1)$	$\mathit{Follow}(r_1) \cup (\mathit{Last}(r_1) \times \mathit{First}(r_1))$

Tablica 1: Feature *Null* and sets *First*, *Last*, and *Follow* for particular types of regular expressions.

State numbers that are targets of **transitions leaving the initial state** are stored in set *First*. The transitions are labeled with symbols associated with the target states.

Final state numbers (except for the initial state, whose finality depends on the feature *Null*) are stored in set *Last*.

Transitions leaving states other than the initial one are stored in set *Follow*. The first item in a pair of states stored in *Follow* determines the source state state of a transition, the second one — the target state.

For more information about construction of automata from regular expressions see the following publications:

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman, *Automata Theory, Languages, and Computation*, Pearsons International Edition, 2007;
- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers. principles, Techniques, and Tools*, Addison Wesley Longman, 1986;
- H.M.M. ten Eikelder, H.P.J. van Geldrop, *On the correctness of some algorithms to generate finite automata for regular expressions*, Department of Mathematics and Computing Science, Eindhoven University of Technology, Computing Science Note 93/32, Eindhoven, September 1993.

4 Skeletal program

A skeletal program for construction of automata from regular expressions is in file `z7c.tgz`. It contains a complete lexical analyzer (scanner) and a partial parser. The parser needs to be completed. The input to the program is the text of a regular expression, where $\Sigma = \{0, \ldots, 9, a, \ldots, z\}$. The output is a file in format of `dot` — a program from `graphviz` package available at http://www.graphviz.org/. On the output, state diagrams of 3 automata are given: a nondeterministic automaton (the result of Yamada-McNaughton-Glushkov construction), deterministic one (Glushkov automaton after determinization), and the minimal deterministic automaton.

Only syntax rules placed between pairs of characters %% in the parser program need to be completed. The skeletal program contains a single rule that recognizes a symbol in the alphabet or an empty sequence of symbols ε. One has to add rules for remaining constructions. The rule that handles a symbol in the alphabet creates a new states calling function `create_state()`. That function is used in no other rule; it is used also in function `create_NFA` for creation of the initial state of the automaton.

Every rule that creates a new automaton for a new subexpression (not every subexpression requires creation of a new automaton) must call function `createRE()` to achieve that. The parameters of that function are: feature *Null*, a set of states (numbered symbols) *First*, a set of states *Last*, and a set of pairs of states *Follow*. The result of the function is a tuple containing the feature *Null* and the sets *First*, *Last*, and *Follow*. The user has no direct access to them. If `x` is a result of `createRE()` invocation, then feature *Null* is extracted using function `RE_NULL(x)`, set *First* — using `RE_FIRST(x)`, *Last* — `RE_LAST(x)`, and *Follow* — `RE_FOLLOW(x)`.

Feature *Null* can take two values: `FALSE` and `TRUE`. And so is the result of function `RE_NULL()`. One can use C operators `&&` and `||` on values of feature *Null*.

Sets *First* and *Last* are initially either empty denoted with constant `EMPTY_FIRST_OR_LAST_SET`, or they are created in a rule for a single symbol (and only there) using function `create_set()` that has a parameter that is a state (a numbered symbol) coming from function `create_state()`. Sets with larger number of members are created with a function `merge_sets()` for adding sets, whose parameters are sets *First* or *Last*.

Sets of pairs of states *Follow* are initially either empty denoted with constant `EMPTY_FOLLOW_SET`, or are created with a Cartesian product function `set_product()` that acts on two sets of states. Sets of pairs of states can be added using function `merge_follow_sets()`.

5 Task to be completed

1. Handle an empty set $\emptyset$ and an expression in parentheses.
2. Handle an alternative.
3. Handle concatenation. Use priority of a virtual CONCAT operator (concatenation means sticking two expressions one after another; no additional operator is necessary). An example of setting priority of an operation using a virtual operator (the lexical analyzer never returns CONCAT value) can be found by invoking `info bison` (or `Ctrl-H i` in emacs), going to `Examples`, and then to `Infix Calc`.
4. Handle transitive closure.
5. Handle an extension: closure operator "+". Here the lexical analyzer needs to be supplemented, and the priority of the operator must be fixed. Note that ε^+ or $((abc)^*(\varepsilon|def))^+$ are correct expressions.

If for the regular expression from the example you get automata different from those on diagrams, try the program on simpler expressions like „01" „0|1" „0*". If they are OK, errors might hide in parameters of function `createRE`. A frequent error is to use $2 instead of $3, where $2 means the "value" of an operator.

6 Example

Expression: **0(0|1)*0** (a sequence of zeros and ones beginning and ending with zero). We have: $0_1(0_2|1_3)^*0_4$, *Null*=false, *First*=$\{0_1\}$, *Last*=$\{0_4\}$, *Follow*=$\{(0_1,0_2),(0_1,1_3),(0_1,0_4),(0_2,0_2),(0_2,1_3),(0_2,0_4),(1_3,0_2),(1_3,1_3),(1_3,0_4)\}$

The result as text:

```
digraph "\"0(0|1)*0\"" {
  rankdir=LR;
  node[shape=circle];

  subgraph "clustern" {
    color=blue;
    n3 [shape=doublecircle];
    n [shape=plaintext, label=""]; // dummy state
    n -> n4; // arc to the start state from nowhere
    n4 -> n0 [label="0"];
    n0 -> n1 [label="0"];
    n0 -> n2 [label="1"];
    n0 -> n3 [label="0"];
    n1 -> n1 [label="0"];
    n1 -> n2 [label="1"];
    n1 -> n3 [label="0"];
    n2 -> n1 [label="0"];
    n2 -> n2 [label="1"];
    n2 -> n3 [label="0"];
    label="NFA"
  }

  subgraph "clusterd" {
    color=blue;
```

```
    d2 [shape=doublecircle];
    d [shape=plaintext, label=""]; // dummy state
    d -> d0; // arc to the start state from nowhere
    d0 -> d1 [label="0"];
    d1 -> d2 [label="0"];
    d1 -> d3 [label="1"];
    d2 -> d2 [label="0"];
    d2 -> d3 [label="1"];
    d3 -> d2 [label="0"];
    d3 -> d3 [label="1"];
    label="DFA"
  }

  subgraph "clusterm" {
    color=blue;
    m0 [shape=doublecircle];
    m [shape=plaintext, label=""]; // dummy state
    m -> m1; // arc to the start state from nowhere
    m0 -> m0 [label="0"];
    m0 -> m2 [label="1"];
    m1 -> m2 [label="0"];
    m2 -> m0 [label="0"];
    m2 -> m2 [label="1"];
    label="min DFA"
  }
}
```

The result as diagrams: