Finite Automata from Regular Expressions — Thompson Construction

1 Aim of the exercise

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

2 Environment

Synopsis of `dot`:

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

To easiest way to get a result is to write e.g.:

```
echo '0(0|1)*0' | ./z7a | dot -Tps > result.ps; gv result.ps &
```

3 Thompson construction

An automaton is constructed from its parts in a way similar to the way an arithmetic expression is evaluated. A characteristic feature of the Thompson construction is that the resulting automaton has not only one initial state, but a **single final state** as well. In an arithmetic expression, the basic building blocks are integer and real numbers as well as memory locations (M1, M2, ...). In a regular expression, the basic building blocks are: an empty set $\emptyset$, an empty sequence ε, and a symbol σ from the alphabet Σ.

An **empty set** $\emptyset$ is equivalent to an automaton with an initial and a final state, but without any transitions. An **empty sequence** ε is equivalent to an automaton with an initial and a final state, and with a transition from the initial to the final state labeled with the empty sequence ε. A symbol from the alphabet $\sigma \in \Sigma$ is equivalent to an automaton with an initial and a final state, and with a transition from the initial to the final state labeled with that symbol.

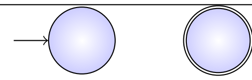
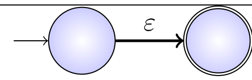
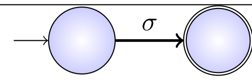
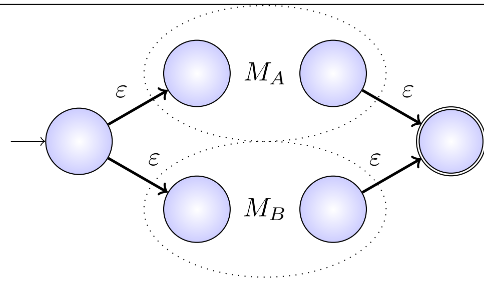
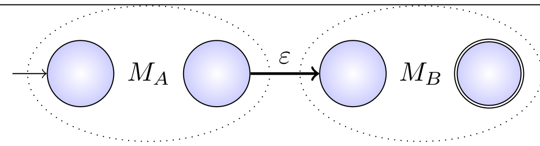
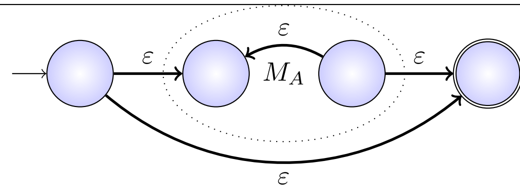
Once one has the basic building blocks, one can create larger automata using concatenation, alternative, and transitive closure. Let M_A and M_B be automata recognizing expressions R_A and R_B respectively. An automaton for **concatenation** R_AR_B of expressions R_A and R_B is constructed by making non-final the final state of the automaton M_A, and linking it with a transition labeled with ε with the initial state of M_B. The initial state of the resulting automaton is the initial state of M_A, and the final state — the final state of M_B.

An automaton for the **alternative** $R_A|R_B$ of expressions R_A and R_B is created by adding a new initial state and a new final state. Four transitions labeled with ε (ε-transitions) are added as well: two from the new initial state to the initial states of M_A and M_B, and two from the final states of M_A and M_B to the new final state. The final states of M_A and M_B stop being final.

An automaton for **transitive closure** R_A^* of a regular expression R_A is constructed by adding a new initial state and a new final state. Four new ε-transitions are also added: from the new initial state to the initial state of M_A, from the new initial state to the new final state, from the final state of M_A to the initial state of M_A, and from the final state of M_A to the new final state. The final state of M_A stops being final.

Parentheses can be used in regular expressions to group items just like they are used in arithmetic expressions. The Thompson construction is summarized in a table on the next page, and additional information about the issues presented here can be found in the following text books:

- 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;

	Thompson
$\emptyset \in RE$	
$\varepsilon \in RE$	
$\sigma \in \Sigma \Rightarrow \sigma \in RE$	
$R_A, R_B \in RE \Rightarrow R_A \mid R_B \in RE$	
$R_A, R_B \in RE \Rightarrow R_A R_B \in RE$	
$R_A \in RE \Rightarrow R_A^* \in RE$	

4 Skeleton program

A skeleton program for construction of automata from regular expressions is provided in file `z7a.tgz`. It contains a full lexical analyzer, 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 a format accepted by `dot` program from the `graphviz` package available from http://www.graphviz.org/. The output contains transition diagrams for three automata: a nondeterministic automaton (the result of the Thompson construction), a deterministic one (Thompson construction result after determinization), and the minimal deterministic automaton.

The only part to be completed is a set of syntax rules between pairs of characters %% in the parser program. Only a rule recognizing a symbol from the alphabet or an empty sequence ε is provided. Rules for all other constructions need to be added. Function `create_state` creates a state. It has no parameters, and it returns the number of the created state. Transitions are created with function `create_transition`. It has three parameters: source state number, target state number, and the transition label. To label a transition with an ε, one must use the constant `EPSILON` as the label.

To construct automata for larger expressions from smaller ones, one must know initial and final state numbers for subexpressions. To avoid problems with returning structures in pure C, the initial state and the final state of an expression are stored in `subREs` table. To store them for the currently recognized expression, one must use `createRE` function. The result of the function is an index of an item in `subREs`. It should be transferred as the result of a syntactic construction in variable $$ at the end of an action for the expression:

```
$$ = createRE(initial,final); /* initial and final state for current RE */
```

To extract the initial state and the final state of an automaton recognizing an expression, one uses FIRST() and LAST() macros, e.g. if one wants to determine the initial state number of an automaton for the third expression in a production rule for a syntactic construction, one should write:

```
FIRST($3) /* for the final state replace ,,FIRST'' with ,,LAST'' */
```

5 Task to be completed

1. Write a rule for ∅ and for an expression in parentheses.
2. Write a rule for alternative.
3. Write a rule for concatenation. Use priority of CONCAT operator (concatenation means gluing two expressions one after another; no additional operator is needed).
4. Write a rule for transitive closure.
5. Write an extension: a closure operator ”+”. One must supplement the lexical analyzer and set the priority.

If for the regular expression from the example below different automata are created than those depicted in figures, one must verify the output of the program using simple expressions like ”01” ”0|1” ”0*”, and compare the results with figures in the table. If they are OK, additional errors might linger in parameters of `createRE` – setting the initial and the final state. Automata can have different state numbers — the result is still correct.

6 Example

Expression: **0(0|1)*0** (a sequence of 0s and 1s starting and ending with 0).

Textual output:

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

  subgraph "clustern" {
    color=blue;
    n11 [shape=doublecircle];
    n [shape=plaintext, label=""]; // dummy state
    n -> n0; // arc to the start state from nowhere
    n0 -> n1 [label="0"];
    n2 -> n3 [label="0"];
    n4 -> n5 [label="1"];
    n6 -> n2 [fontname="Symbol", label="e"];
    n6 -> n4 [fontname="Symbol", label="e"];
    n3 -> n7 [fontname="Symbol", label="e"];
    n5 -> n7 [fontname="Symbol", label="e"];
    n7 -> n6 [fontname="Symbol", label="e"];
    n8 -> n6 [fontname="Symbol", label="e"];
    n7 -> n9 [fontname="Symbol", label="e"];
    n8 -> n9 [fontname="Symbol", label="e"];
    n10 -> n11 [label="0"];
    n9 -> n10 [fontname="Symbol", label="e"];
    n1 -> n8 [fontname="Symbol", label="e"];
    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"
  }
}
```

Diagrams: