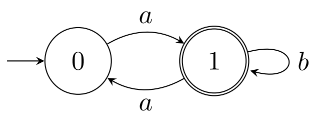
Algorithms and Data Structures 2014
Exercises and Solutions Week 11

1 Converting DFA to regular expressions

In this exercise you will explore a technique for converting deterministic finite automata (DFA) to regular expressions. The idea is based on Kleene's proof of the equivalence of finite automata and regular expressions (e.g., see the course 'Talen en Automaten').

DFA are often represented by directed graphs called (state) transition diagrams. The vertices (denoted by single circles) of a transition diagram represent the states of the DFA and the edges labeled with an input symbol correspond to the transitions. An edge (p, q) from vertex p to vertex q with label σ represents the transition $(p, \sigma) = q$. The accepting states are indicated by double circles whereas the intial state is marked with an incoming arrow (not originating from a vertex).

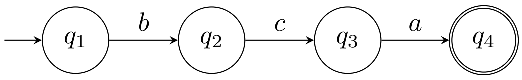
The following figure shows an example of a DFA:



A minimal regular expression for representing this automaton is $a(b|aa)^*$.

Transitive Closure Method

Suppose the DFA given below is to be represented as a regular expression.



Notice that the regular expression for the transition from q_1 to q_2 is b, the transition from q_2 to q_3 is c and so on. Furthermore, the regular expression representing the transition from q_1 to q_3 is the concatenation of the regular expressions b and c being bc.

Likewise, we can find the regular expression for the complete automaton to be bca, since this expression is the concatenation of all transitions from the starting state q_1 to the final state q_4.

More generally, for a path from q_s to q_f, the concatenation of the regular expression for each transition in the path forms a regular expression that represents the same language as the path from q_s to q_f in the automaton itself.

In our simple automaton there exists only one path from the initial to the final state. The situation becomes more complex if states have more than one outgoing egdes.

The first example DFA containing multiple paths from q_0 to q_1 cannot be represented by a simple regular expression. By using the alternation and closure operations, however, it appears that we can refine our previous approach to specify a construction that works for all kinds of DFAs. The essence of this procedure is to determine a transitive closure, similar in nature to the all-pairs shortest path algorithm as given by Floyd-Warshall.

Suppose R^k_{ij} represents the regular expression that corresponds to the path in the DFA from q_i to q_j that does not go via any state higher than q_k. We can construct the final regular expression R_{ij} first by creating R^0_{ij}, and subsequently by constructing R^1_{ij}, R^2_{ij}, ..., $R^{N-1}_{ij} = R_{ij}$. The k^{th} expression R^k_{ij} is recursively defined as:

$$R^k_{ij} = R^{k-1}_{ij} \mid R^{k-1}_{ik} \, (R^{k-1}_{kk})^* \, R^{k-1}_{kj}$$

assuming we have initialized R^k_{ij} as follows.

$$R^0_{ij} \quad = \quad \begin{cases} a & \text{if } i \neq j \text{ and there exists an edge } (q_i, q_j) \text{ with label } a \\ b \,|\, \varepsilon & \text{if } i = j \text{ and there exists an edge } (q_i, q_i) \text{ with label } b \\ \varepsilon & \text{if } i = j \text{ and there exists no edge } (q_i, q_i) \\ \emptyset & \text{otherwise} \end{cases}$$

Here, $\emptyset$ is used to represent an RE that corresponds to the empty language.

In order to limit the size of the resulting RE we can use the following identities as simplification rules (where α denotes an arbitrary RE, and a an arbitrary symbol).

$$\begin{aligned} \alpha\,\emptyset &= \emptyset = \emptyset\,\alpha \\ \alpha \,|\, \emptyset &= \alpha = \emptyset \,|\, \alpha \\ (\varepsilon \,|\, \alpha)^* &= (\alpha)^* \\ \varepsilon \,|\, (\alpha)^* &= (\alpha)^* \\ (a)^* &= a^* \\ \varepsilon^* &= \varepsilon \end{aligned}$$

Remark: $(a)^*$ denotes the regular expression that consists of a single symbol a.

1. Define a function *kleene* that yields the closure for a given RE.
2. Define a function *alternate* yielding the union for two given REs.
3. Define a function *concat* for concatenating two REs.

Apply in each of the functions the simplification rules to keep the size of the result as small as possible.

4. Give an algorithm to compute the transitive closure of a DFA, specified by an $N \times N$ connection matrix.

Solution In pseudocode we can regard the different syntactic constructions for regular expressions as datatype constructors and match on them. This makes the implementation very straightforward. In a language like Java, an approach that is similar to this would be to create subclasses for all types of expressions and then distinguish on the class of the actual expressions passed to the functions.

1.
```
function KLEENE(r)
    if r = ε then
        return ε
    else if r = ε | α then
        return (α)*
    else
        return (r)*
```

   The condition $r = \varepsilon \,|\, \alpha$ expresses two checks: the expression r must be a union and its left subexpression must be ε; the right subexpression can be an arbitrary regular expression, which we name α.

   Note that parentheses are not represented explicitly, so the identity $(a)^* = a^*$ does not mean anything.

2.
```
function ALTERNATE(r, s)
    if r = ∅ then
        return s
    else if s = ∅ then
```

```
        return r
    else if r = ε and s = (α)* then
        return s
    else
        return r | s
```

3.
```
function CONCAT(r, s)
    if r = ∅ or s = ∅ then
        return ∅
    else
        return r s
```

4. We assume the DFA to be specified using an adjacency function assigning to each state i the state j (we identify states with their indices here) that is adjacent to it; it is undefined if there is no such state. The labels of these transitions are given by the label function l. Note that these can also be seen as regular expressions.

```
function CONNECTIONMATRIX(Adj, l)
    for i from 0 to N − 1 do
        for j from 0 to N − 1 do
            if i = j then
                if Adj[i] = j then
                    R_ij ← ε | l(i, j)
                else
                    R_ij ← ε
            else
                if Adj[i] = j then
                    R_ij ← l(i, j)
                else
                    R_ij ← ∅
    for k from 0 to N − 1 do
        for i from 0 to N − 1 do
            for j from 0 to N − 1 do
                R_ij ← ALTERNATE(R_ij, CONCAT(R_ik, CONCAT(KLEENE(R_kk), R_kj)))
    return R
```

Because of way R^0 is initialized, this actually describes the transitive and reflexive closure of the transition function. Note that the superscripts of R have been removed for reasons we saw last week (to be honest, I removed them because the line did not fit). The result of the function describes for each pair of states (i, j) the paths from i to j by the regular expression R_{ij}. From this result, the language of the automaton is given by the alternation of all R_{ij} such that i is initial and j final. The matrix can be transformed into an ordinary boolean connection matrix by checking for all i and j whether $R_{ij} = \emptyset$ (no connection) or not (there is a path from i to j).

2 Simplifying logical expressions

In this exercise we consider logical expressions that are built up from *variables* and the logical operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies) and $\neg$ (not). The first operators are binary; the last unary. The syntax of this language can be described by the following grammar

$$\mathit{Exp} = \mathbf{var} \mid \neg\mathit{Exp} \mid \mathit{Exp} \wedge \mathit{Exp} \mid \mathit{Exp} \vee \mathit{Exp} \mid \mathit{Exp} \Rightarrow \mathit{Exp}$$

As usual, we assume that these operations have different priorities: $\wedge$ binds more tightly than $\vee$ which binds more tightly than $\Rightarrow$. Negation ($\neg$) has the highest priority.

1. Convert the given grammar in such a way that these precedence rules are taken into account.
2. Develop a recursive descent parser for this grammar, first by introducing appropriate datastructures for representing the abstract syntax tree, and subsequently, by defining a set of mutual recursive parsing methods.
3. By assigning concrete (boolean) values to the variables, we can evaluate a logical expression. We call such an assignment an *valuation*. Implement such an evaluator preferably by using the *visitor pattern*. The obvious way to represent valuations is by using *Map*s.

Solution

1.

$$
\begin{aligned}
E &= F \mid F \Rightarrow E \\
F &= G \mid G \vee F \\
G &= H \mid H \wedge G \\
H &= I \mid \neg H \\
I &= \textbf{var}
\end{aligned}
$$

We say that a logical expression e is *valid* if the evaluation of e yields *true* for all valuations. Hence, if e is invalid then there must exist a valuation that, when applied to e, causes e to evaluate to *false*. We call such an valuation a *counter example*. One way to check the validity of an expression is by rigorously trying all possible valuations. The complexity of such a brute force method is 2^N, where N is the number of different variables occurring in e. In the course 'Beweren en Bewijzen' you've seen a more efficient procedure to determine the validity of expressions, based on the so-called *sequent-calculus*. In this calculus, logical formulae are represented as *sequents* which can de reduced (simplified) using a set of predefined reduction rules. If you've forgotten how this procedure works, it is probably wise to reread the corresponding chapters of the textbook. Those who don't have the textbook anymore can use Jaspar's lecture notes which are available via a link placed on the A&D website.

4. Give an algorithm for reducing an initial logical expression until either it can be concluded that the expression is valid or a counter example has been found.