Generating Languages from Non-deterministic Finite Automata (Haskell+Scheme, 7+7 points)

Non-deterministic Finite Automaton

In the seminars, we have encountered *Deterministic Finite Automata (DFAs)*. In this task, we will work with a generalized version of the DFA that is called *Non-deterministic Finite Automaton (NFA)*. NFA is the 5-tuple

– set of states $\mathcal{Q}$,
– a finite set of input symbols Σ (called alphabet),
– a set of transitions $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$,
– a start state q_0,
– a set of final states $\mathcal{F} \subseteq \mathcal{Q}$.

In other words, NFA is just a directed graph whose vertices are states and transitions are edges labelled by symbols from Σ, i.e., if $(s, a, t) \in \Delta$ then there is an edge leading from the state s to the state t labelled by $a \in \Sigma$. We say that NFA accepts a word $w = a_1a_2 \dots a_n \in \Sigma^*$ (i.e., a finite sequence of symbols form Σ) if there exists a path leading from the start state into a final one labelled consecutively by symbols $a_1, a_2, \dots, a_n$.

An example of an NFA is depicted in Figure 1. This NFA accepts e.g. words *abb* or *aba*. On the other hand, it accept neither *ba* nor *abab*.

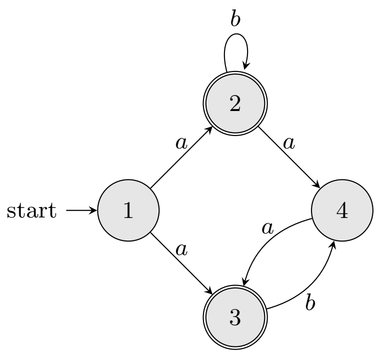

Fig. 1. Example of NFA where $\mathcal{Q} = \{1, 2, 3, 4\}$, $\Sigma = \{a, b\}$, $q_0 = 1$, $\mathcal{F} = \{2, 3\}$ and $\Delta = \{(1, a, 2), (2, b, 2), (1, a, 3), (3, b, 4), (4, a, 3), (2, a, 4)\}$.

Task 4 (Haskell)

Your assignment is to implement a function that generates the language accepted by a given NFA. However, since such a language is potentially infinite, the problem is simplified to listing all words of given length that are accepted by the automaton.

For the purpose of this task, the NFA is defined as follows. First, each possible transition is represented as the triplet

```haskell
data Transition a b = Tr a b a
```

where type `a` represents states in $\mathcal{Q}$ and `b` are the symbols Σ; hence, the transition from the first member of the triplet to the third member of the triplet is possible if and only if the next input symbol is equal to the second member of the triplet. The NFA itself is defined as

```
data Automaton a b = NFA [Transition a b] a [a]
```

where the first member is the exhaustive list of transitions, the second member is the start state q_0, and the third is the list of final states $\mathcal{F}$. The particular automaton from Figs. 1 is defined as follows.

```
nfa::Automaton Int Char
nfa = NFA [Tr 1 'a' 2,
           Tr 2 'b' 2,
           Tr 1 'a' 3,
           Tr 3 'b' 4,
           Tr 4 'a' 3,
           Tr 2 'a' 4]
           1
           [2,3]
```

The first part of the task is to decide whether a particular word is accepted by a given automaton. To that end, implement the function

```
accepts :: (Eq a, Eq b) => Automaton a b -> [b] -> Bool
```

which takes and automaton and a list of symbols that represents the word, and returns `True` iff the word is accepted. Notice, `a` and `b` are instances of `Eq`. The function is used as follows.

```
> accepts nfa "aba"
True
> accepts nfa "abab"
False
```

Hint: One possibility how to implement the function `accepts` is to maintain a list of accessible states after consuming an initial segment of the input word. Consider the NFA from Figure 1 and the word `"aba"`. We start with the list containing the start state `[1]`. Then we modify this list by the transitions corresponding to the letters as follows: `[1] -a-> [2,3] -b-> [2,4] -a-> [3,4]`. Finally, we check if the resulting list of states contains a final state. As `3` is a final state, the output is `True`. Another example for `"abab"` is `[1] -a-> [2,3] -b-> [2,4] -a-> [3,4] -b-> [4]`. As `4` is not a final state, the ouput is `False`. One more example for `"ba"`: `[1] -b-> [] -a-> []`. So output is `False`.

Next, given the automaton, its alphabet, and a length, generate the list of all words of the given length that are accepted by the automaton. Implement the function

```
lwords :: (Eq a, Eq b) => [b] -> Automaton a b -> Int -> [[b]]
```

where the first input is the alphabet as list of unique symbols, the second is the automaton, and the third is the word length. The function returns a list of the accepted words, i.e., a list of lists of symbols. In terms of the task evaluation, the ordering of the words is not relevant. The function operates as follows.

```
> lwords "ab" nfa 0
[]
> lwords "ab" nfa 1
["a"]
> lwords "ab" nfa 3
["aaa","aba","abb"]
```

Hint: first, generate all possible words of the given length. Then, filter the words using the function `accepts`.

Task 3 (Scheme)

You task is the same as in Task 4, i.e., implement the functions `accepts` and `lwords`. In this task, you may assume that the input word will always be a string. Thus you can use functions `string->list` and `list->string` for converting strings to lists of characters and vice versa. Of course, we need to adapt the automaton representation for Scheme. First, each possible transition is represented as the triplet:

```
(define (make-trans from_state symbol to_state)
  (list from_state symbol to_state))

; Accessor functions
(define get-from_state car)
(define get-symbol cadr)
(define get-to_state caddr)
```

where `from_state` and `to_state` are states in $\mathcal{Q}$ and `symbol` is a symbol in Σ. We can construct the NFA itself as

```
(define (make-automaton trans init_state final_states)
  (list trans init_state final_states))

; Accessor functions
(define get-trans car)
(define get-init_state cadr)
(define get-final_states caddr)
```

where the first member `trans` is the exhaustive list of transitions, the second member `init_state` is the start state q_0, and the third member `final_states` is the list of final states $\mathcal{F}$. The particular automaton from Figures 1 is defined as follows.

```
(define nfa
  (make-automaton
   (list (make-trans 1 #\a 2)
         (make-trans 2 #\b 2)
         (make-trans 1 #\a 3)
         (make-trans 3 #\b 4)
         (make-trans 4 #\a 3)
         (make-trans 2 #\a 4))
   1
   (list 2 3)))
```

Implement in Scheme the function

```
(accepts automaton word)
```

which takes an automaton and a string that represents the word to be parsed by the automaton, and returns `#t` if the word is accepted, `#f` otherwise. The function is used as follows:

```
> (accepts nfa "aba")
#t
> (accepts nfa "abab")
#f
```

Next, given the automaton, its alphabet, and a length, generate the list of all words of the given length that are accepted by the automaton. Implement the function

```
(lwords alphabet automaton n)
```

where the first input `alphabet` is the alphabet passed as a string, the second input `automaton` is the automaton, and the third input `n` is the word length. The function returns a list of the accepted words, i.e., a list of lists of characters. In terms of the task evaluation, the ordering of the words is not relevant. The function operates as follows:

```
> (lwords "ab" nfa 0)
'()
> (lwords "ab" nfa 1)
'("a")
> (lwords "ab" nfa 3)
'("aaa" "aba" "abb")
```

For testing purposes your file should be named `task3.rkt` and start with the following lines:

```
#lang racket
(provide accepts
         lwords)
```