Introduction

In the previous tutorial you worked with finite state machines (FSM) implemented in Haskell. An FSM is a finite state machine with five components

```
--------------- FSM states symbols transitions starting accepting ------------------
data FSM q = FSM (Set q) (Set Sym) (Set(Trans q)) (Set q) (Set q) deriving Show
mkFSM :: Ord q => [q] -> [Sym] -> [Trans q] -> [q] -> [q] -> FSM q
mkFSM qs as ts ss fs =  -- a convenience function constructing FSM from lists
  FSM (fromList qs) (fromList as) (fromList ts) (fromList ss) (fromList fs)
```

Some FSA are deterministic

```
isDFA :: Ord q => FSM q -> Bool
isDFA (FSM qs as ts ss fs) = (size ss == 1)
  && and[ length[ q' | q' <- toList qs, (q, a, q')`member`ts ] == 1
        | q <- toList qs, a <- toList as ]
```

In this tutorial we work with machines that may include ε-transitions. We call these NFA (non-deterministic finite-state automata). The declaration is like that for `FSM` , but with an extra component, the set of ε-transitions. We provide a convenience function to create a `NFA` from six lists.

```
------- NFA states symbols transitions e-transitions starting accepting -------------
data NFA q = NFA(Set q)(Set Sym)(Set(Trans q))(Set (q,q))(Set q)(Set q) deriving Show
mkNFA :: Ord q => [q] -> [Sym] -> [(q, Sym, q)] -> [(q, q)] -> [q] -> [q] -> NFA q
mkNFA qs as ts es ss fs =  -- a convenience function constructing NFA from lists
  NFA (fromList qs)(fromList as)(fromList ts)(fromList es)(fromList ss)(fromList fs)
```

We will not allow any reflexive ε-transitions.

We can convert any FSM to an NFA, trivially, by giving it an empty list of ε-transitions – and if an NFA has no ε-transitions we can derive an FSM.

```
---------------------- conversion from FSM to NFA ---------------------------------
asNFA :: FSM q -> NFA q
asNFA (FSM qs as ts ss fs) = NFA qs as ts empty ss fs
-- converting an NFA with no e-transitions to an FSM
asFSM :: NFA q -> FSM q
asFSM  (NFA qs as ts es ss fs)
  | null es = FSM qs as ts ss fs
  | otherwise = error "has e-transitions"
```

We've included the solutions from last week in the template for this tutorial, with some functions renamed to make it clear that they apply to FSM. For example, the function to convert from an arbitrary FSM to a DFA appears as:

```
fromFSMtoDFA :: (Ord q) => FSM q -> FSM (Set q)
```

Your first exercise for this tutorial is to provide the key function needed to complete the corresponding function for `NFA` .

```
fromNFAtoDFA :: (Ord q) => NFA q -> NFA (Set q)
```

ε-closure

An NFA accepts a word w iff there is a path along labelled transitions and ε-transitions from a starting state to an accepting state, such that the labelled transitions spell out the word, w.

For FSM we considered transitions between sets of states, which we called superstates. For NFA our superstates will be ε-closed sets of states.

We say a set `qq` of states of a DFA is ε-closed iff whenever `q` $\in$ `qq` and there is an ε-transition from `q` to `q'`, then `q'` $\in$ `qq`.

1. **Mandatory** The function `eStep es qq` (provided in the template) applies the ε-transitions in `es` to the states in `qq` to return the set of states reachable from `qq` by a single ε-step.

   ```
   eStep  :: Ord q =>  Set (q,q) -> Set q -> Set q
   eStep es qq = fromList [ q' | (q, q') <- toList es, q `member` qq ]
   ```

   Your first task is to replace `undefined` in the code below, to complete the definition of `eClose` : `eClose es qq` should return the set of states reachable in any sequence (including the empty sequence) of ε-steps from `qq`.

   ```
   eClose :: Ord q => Set(q,q) -> Set q -> Set q
   eClose es qq =
     let new = eStep es qq \\ qq
     in if null new then undefined else eClose es undefined
   ```

   Hint: Look at the code for `reachableFSM` and `reachableNFA`.

The code for the following functions, which are provided, follows closely the pattern we used for the FSM versions.

```
ddeltaNFA :: (Ord q) => NFA q -> (Set q) -> Sym -> (Set q)
acceptsNFA :: (Ord q) => NFA q -> [Sym] -> Bool
nextNFA :: (Ord q) => NFA q -> Set(Set q) -> Set(Set q)
reachableNFA :: (Ord q) => NFA q ->  Set(Set q) -> Set(Set q)
dfinalNFA :: (Ord q) => NFA q -> Set(Set q) -> Set(Set q)
dtransNFA :: (Ord q) => NFA q -> Set(Set q) -> Set(Trans (Set q))
```

The only difference is in `ddeltaNFA`, which uses `eClose` to return an ε-closed superstate.

```
ddeltaNFA :: (Ord q) => NFA q -> (Set q) -> Sym -> (Set q)
ddeltaNFA (NFA  qs as ts es ss fs ) source  sym =
  eClose es (transition ts source sym)
```

Compare this with `ddeltaFSM`.

2. The function `fromNFAtoDFA` produces a DFA equivalent for a given NFA — that is to say, it produces an equivalent NFA that has no ε-transitions, and exactly one transition of the form (q, s, q') for each state, symbol pair, q, s —

   ```
   fromNFAtoDFA :: (Ord q) => NFA q -> NFA (Set q)
   ```

   Mandatory How does the implementation of this function differ from the implementation of `fromFSMtoDFA` ?

You are strongly advised to practice some by-hand examples of the `fromNFAtoDFA` conversion. You can find many examples in past papers, and in the FSM workbench. There are some **mandatory** examples in Q10 of this tutorial.

In the constructions that follow, the types of the states can become quite involved – sets of sets of sets of states. It is sometimes helpful to forget all this structure and relabel the states with numbers. Two convenience functions are provided to do this:

```
intFSM :: (Ord q) => FSM q -> FSM Int
intNFA :: (Ord q) => NFA q -> NFA Int
```

The work you have done so far allows you to convert any NFA to a DFA – but with no further work you can actually do better. The functions you have implemented are sufficient to produce a minimal DFA (we won't prove that in this course).

```
minimalDFA :: Ord q => NFA q -> FSM Int
minimalDFA = asFSM.intNFA.fromNFAtoDFA.reverseNFA.fromNFAtoDFA.reverseNFA
```

You can then use `tidyFSM` to present your machine using the black-hole convention.

The template includes the following functions to tidy up your machines for human consumption. In presenting a machine, it helps to omit unreachable states, the function `pruneFSM` does this for FSM.

```
pruneFSM :: Ord q => FSM q -> FSM q
```

As we've seen with the black hole convention it often makes things even simpler if we omit states from which we cannot reach any accepting state. We can remove both these and the unreachable stets with the following function.

```
tidyFSM :: Ord q => FSM q -> FSM Int
tidyFSM = intFSM . reverseFSM . pruneFSM . reverseFSM . pruneFSM
```

The remaining states are called the *live states* of the machine.

regex → NFA

The main business of this tutorial is for you to implement some of the constructions described in Chapter 30 *Regex to Machines*.

In this section you will implement functions that allow you to create an FSM for any regex. For these questions, we use alphabets that are subsets of the lower-case letters `['a'..'z']`. When we combine two machines we take the union of their alphabets.

You will use a variant of Thompson's construction[1] to build an NFA that recognises the language defined by a given regular expression. This construction is also described in Ch. 30 of the textbook.

The key idea of Thompson's construction is to keep it simple – each NFA we construct will have a single start state and a single accepting state. The machine must have no transitions ending in its start state, and no transitions starting from its accepting state. We call such a machine a Thompson NFA. Here's a function that tests for this property.

```
isThompson :: Eq a => NFA a -> Bool
isThompson (NFA qs as ts es ss fs) = case (toList ss,toList fs) of
  ([s],[f]) -> (not . or) [ q'==s||q==f | (q,_,q') <- ts] &&
               (not . or) [ q'==s||q==f | (q,q') <- es]
  _ -> False
```

The machines constructed below may assume this property for any NFA arguments; under this assumption they must guarantee this property for the NFA they return.

Basic regex We begin by asking you to construct some basic building blocks - machines that recognise the languages corresponding to some simple patterns.

3. **Mandatory** Replace occurrences of `undefined` to give a **Thompson** NFA for each of the following questions.

   (a) Write a function `stringNFA :: String -> NFA Int` that given a string `xs` returns a Thompson NFA that accepts only the string `xs`.

   ```
   stringNFA :: String -> NFA Int
   stringNFA xs = mkNFA qs as ts es ss fs where
     qs = [0..n]
     as = xs
     ts = undefined
     es = []
     ss = [0]
     fs = [n]
     n = undefined
   ```

[1] Diagrams are taken from this Wikipedia article under a creative commons licence; follow the link for details. In the diagrams, N(s) and N(t) are the NFA of regular expressions s and t, respectively.

(b) Define a value `nullNFA :: NFA ()` that represents a Thompson NFA that accepts the empty language $\varnothing$, with no strings. Hint: This must satisfy the Thompson conditions so it should include a single start state and a single accepting state.

```
nullNFA :: NFA Bool  -- NB ensure this is Thompson
nullNfa = undefined
```

(c) Define a value `dotNFA :: NFA Bool` that represents a Thompson NFA that accepts every single-character string from our alphabet `['a'..'z']]` , and no other strings.

```
dotNFA :: NFA Bool
dotNFA = mkNFA qs as ts es ss fs where
    qs = [False,True]
    as = ['a'..'z']
    ts = undefined
    es = []
    ss = [False]
    fs = [True]
```

Constructing Machines Understanding the constructions used in this section is **mandatory**; but completion of some parts of the Haskell code is optional.

To get you started, we give one construction as an example. We can convert any NFA to Thompson form by adding two new states and some ε-transitions. We use this example to introduce some new ideas.

To add the new states, we will call them `Q` and `F` , we use a data declaration:

```
data QF q = Q | E q | F deriving (Eq, Ord,Show)
```

A value of type `QF q` is either an existing q, labelled `E q`, or one of the new values, `Q,F`.

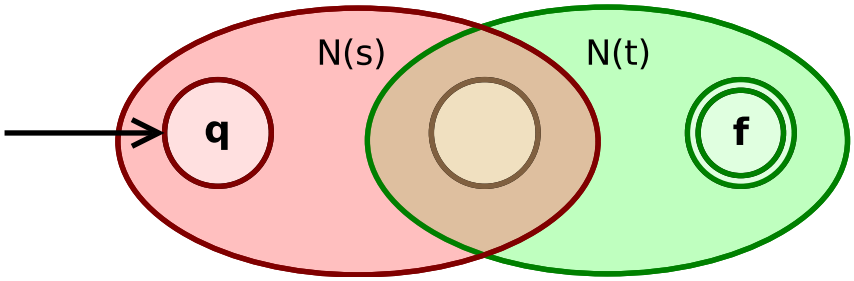
We will use the following functions to map the labelled transitions and ε-transitions of the original NFA into this new type.

```
mapTrans :: (Ord a, Ord b) => (a -> b) -> Set(Trans a) -> Set(Trans b)
mapTrans f = mapS (\(q, a, q') -> (f q, a, f q'))

mapEs :: (Ord a, Ord b) => (a -> b) -> Set (a, a) -> Set (b, b)
mapEs f = mapS (\ (q, q')-> (f q, f q'))

thompson :: Ord q => NFA q -> NFA (QF q)
thompson (NFA qs as ts es ss fs) =
  NFA qs' as ts' es' ss' fs' where
   qs' = ss' \/ fs' \/ mapS E qs
   ts' = mapTrans E ts
   es' = mapEs E es
         \/
         fromList [ (Q, E q) | q <- toList ss ]
         \/
         mapS (\q -> (E q, F)) fs
   ss' = singleton Q
   fs' = singleton F
```

In the definition of `es'` , we first map the ε-transitions from the original machine into the `QF q` type; then add a transition from `Q` to each embedded start state (`fromList [ (Q, E q) | q <- toList ss ]`); then add a transition to `F` from each embedded accepting state (`mapS (q-> (E q, F)) fs`). You can use either of these styles: converting from and to lists to use a comprehension, or using `mapS` . You will need to use these ideas again below.

4. **Mandatory** Draw a diagram showing the construction performed by `thompson` . Your diagram should illustrate the interpretations of `Q` , `F` , and `E` ,

Thompson's construction for concatenation is described by the following diagram.



We will be lazy and use the following construction instead

```
concatNFA :: (Ord a, Ord b) => NFA a -> NFA b -> NFA (Either a b)
concatNFA (NFA qs as ts es ss fs)(NFA qs' as' ts' es' ss' fs') =
  NFA qs'' as'' ts'' es'' ss'' fs'' where
    qs'' = mapS Left qs \/ mapS Right qs' -- = disjointUnion qs qs'
    as'' = as \/ as'
    ts'' = mapTrans Left ts \/ mapTrans Right ts'
    es'' = mapEs Left es \/ mapEs Right es'
           \/
           cartesianProduct (mapS Left fs) (mapS Right ss')
    ss'' = mapS Left  ss
    fs'' = mapS Right fs'
```

In this example we use the type `Either a b` to represent the disjoint union of the states of the two FSM. This includes things of type `a` with a label `Left` and things of type `b` with label `Right` . This allows us to model two machines, side-by-side. The `Data.Set` library provides a function

```
disjointUnion :: Set a -> Set b -> Set (Either a b)
disjointUnion xs ys = map Left xs `union` map Right ys
```

So, as per the comment, we could have used `disjointUnion` to form `qs''` .

The function, `cartesianProduct` , from `Data.Set` , forms the set of pairs drawn from two sets of things.

```
cartesianProduct :: Set a -> Set b -> Set (a, b)
```

We use this to define some new ε-transitions.

Provided its arguments are Thompson this construction returns a Thompson NFA that recognises the concatenation of the languages of its two arguments.

5. **Mandatory** Draw a diagram describing the construction performed by `concatNFA` . Explain why the concatenation of two Thompson machines, using this function, is a Thompson machine.

6. The union of two NFAs represents the two machines placed beside each other, running in parallel. The start/accepting states are just the disjoint unions of the start/accepting states of the two machines. The labelled- and ε-transitions are just mapped from the original machines. The alphabet is the union of the two alphabets.

   Optional Complete the following definition of this construction.

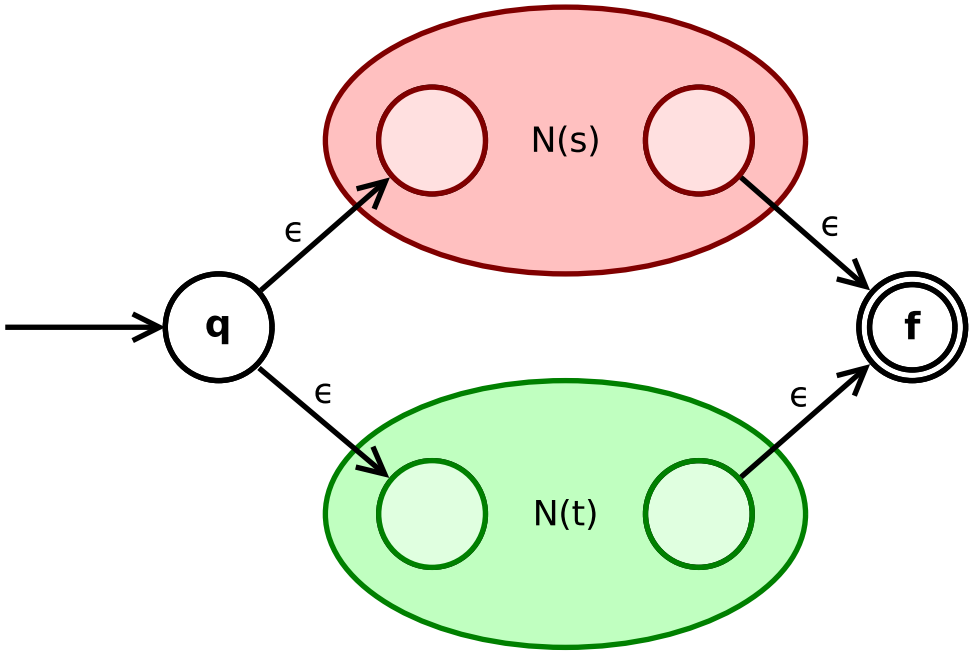
   ```
   unionNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (Either q q')
   unionNFA (NFA qs as ts es ss fs) (NFA qs' as' ts' es' ss' fs') =
     NFA qs'' as'' ts'' es'' ss'' fs'' where
     qs'' = disjointUnion qs qs'
   ```

```
as'' = as \/ as'
ts'' = undefined
es'' = mapEs Left es \/ mapEs Right es'
ss'' = disjointUnion ss ss'
fs'' = undefined
```

To complete the construction of a Thompson union is now simple (using an earlier function).

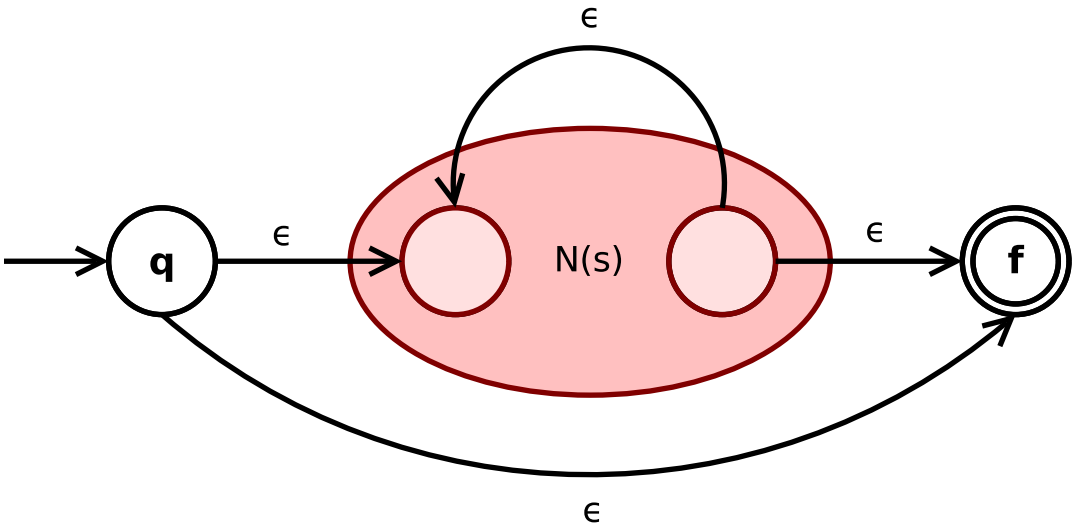
7. **Optional** Complete the following declaration to match the picture.

```
tUnionNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (QF (Either q q'))
tUnionNFA m n = undefined
```



8. The Kleene star closure of an NFA, `A`, accepts any input word which consists of a concatenation of words accepted by `A`. This includes the empty concatenation: the empty string is accepted.

   For this question you will implement the construction illustrated



Optional We use the type `QF q` introduced above. Replace the `undefined`s to complete the function so that it returns the Kleene star closure of the input automaton. Your function should work on any Thompson NFA.

```
tStarNFA :: Ord q => NFA q -> NFA (QF q)
tStarNFA (NFA qs as ts es ss fs) =
  NFA qs' as ts' es' ss' fs'
  where qs' = ss'\/fs'\/ mapS E qs
        ts' = undefined
        es' = fromList [ undefined,  (Q, E (the ss)), (E (the fs), F)
                       , (E (the fs), E (the ss)) ]
              \/
              mapEs E es
        ss' = singleton Q
        fs' = undefined
        the xs = case toList xs of
                   [x] -> x
                   _   -> error "no unique the in tStarNFA"
```

Regular expressions We use the following data type to represents regex (`Regex0` corresponds to the empty language, ∅):

```
data Regex =
  S String | Regex0 | Dot
  | (:|:) Regex Regex -- alternation
  | (:>:) Regex Regex -- concatenation
  | Star Regex        -- Kleene-star
  deriving Show
```

9. **Optional** Implement the function `regex2nfa` so that it produces an NFA that recognises a given regex – your function should use the thompson construction.

```
regex2nfa :: Regex -> NFA Int
regex2nfa (S s)     = undefined
regex2nfa Regex0    = undefined
regex2nfa Dot       = undefined
regex2nfa (r :|: s) = undefined  (regex2nfa r)(regex2nfa s)
regex2nfa (r :>: s) = undefined  (regex2nfa r)(regex2nfa s)
regex2nfa (Star r)  = undefined  (regex2nfa r)
```

Composing `tidyFSM . minimalDFA . regex2nfa` gives a function that takes a regex and produces a minimal tidied machine.

10. **Mandatory** You should complete this exercise by hand, even if you have not completed the Haskell implementation.

    For each example you should write out a derivation or a corresponding DFA, by hand, showing the NFA corresponding to the regex, and the DFA derived using the power-set construction.

    Optional If you have completed the implementation you should use your implementation to derive minimal tidied machines for the regex, and compare these with those you have derived by hand.

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

    (b) `(a|(ab)*)*`

    (c) `∅*`

11. **Optional** How many live states are there in the minimal dfa that recognises the language consisting of the words, all converted to lower case, in this sentence?

    Hint: Your code might look like this:

```
sentence = "how many states are there in the minimal dfa that recognises the"
  ++" language consisting of the words in this sentence all in lower case"

(tidyFSM.minimalDFA.regex2nfa)(foldr undefined undefined $ map S $ words sentence)
```

12. **Optional** Use the Haskell tools developed in this tutorial to solve Q1b of CL tutorial 9.