NFA[1]

In this tutorial you will work with FSM, DFA, and NFA. You will build and run these in Haskell. Some questions overlap with some of the optional material from tutorial FP9.

We use the library `Data.Set` to represent sets.

```
import Data.Set(Set, insert, empty, member, toList)
import qualified Data.Set as Set
```

and introduce the following infix operators, for convenience and readabiity:

```
(\/) :: Ord a => Set a -> Set a -> Set a
(\/) = Set.union
(/\) :: Ord a => Set a -> Set a -> Set a
(/\) = Set.intersection
(\\) :: Ord a => Set a -> Set a -> Set a
(\\) = Set.difference
mapS :: Ord b => (a -> b) -> Set a -> Set b
mapS = Set.map
set :: Ord q => [q] -> Set q
set = Set.fromList
```

Make sure you understand the various functions defined on sets mentioned above. We use `set = Set.fromList` and `fromList` to convert lists to sets and *vice-versa*.

Before getting to grips with machines, we start with a short exercise on reachability.

Reachability

This idea of reachability is very general – you saw it in our foaf (friend-of-a-friend) computations with a social graph. Everyone in the network was reachable by the foaf-oaf... relation. Reachability[2]crops up in many areas of informatics.

We will define a function[3]

```
reach :: Ord p =>  (Set p -> Set p) -> (Set p -> Set p)
```

The first argument of `reach` is a function `step :: Set p -> Set p` such that `step ss` returns the set of states we can reach *in one step* from the states (or people, or nodes, or whatever, depending on your application) in `ss`. The value of `reach step :: Set p -> Set p` is another function of the same type, but this function tells us which states we can reach eventually if we are prepared to keep going, for as long as it takes.

Using a pattern you saw in FP9 we can define a version of this function:

```
reach' :: Ord q => (Set q -> Set q) -> Set q -> Set q
reach' step gs = if gs' == gs then gs else reach' step gs'
  where gs' = gs \/ step gs
```

Of course for some `step` functions this computation might never terminate. However, if we work within a finite set of nodes the computation will terminate.

[1]This tutorial is adapted from an earlier tutorial authored by Philip Wadler and others.

[2]A phrase highlighted like this is is a live link – in this case to a relevant Wikipedia article.

[3]The second pair of parentheses in the type declaration is redundant – we would normally write `reach :: Ord p => (Set p -> Set p) -> Set p -> Set p`, which means exacly the same thing – we include the parentheses here to stress the fact that we will normally use `reach` as an operator that takes a function as argument and returns a function as result.

reach 2.0 We have called our function `reach'` since your next task is to write a more efficient version, which we will call `reach`. To see that there are opportunities for improvement, try evaluating, the behaviour of `reach'` by hand using the function `succs`, which returns the set of successors, mod 10, of the numbers in its argument list:

```
succs :: Set Int -> Set Int
succs xs = mapS ((`mod`10).(+1)) xs
```

For example, to evaluate `reach succs (set[0])`, reach computes the following, in succession,

```
> succs (set[0])
fromList [1] -- this is how a Set is Shown, as fromList xs
> succs (set [0,1])
fromList [1,2]
> succs (set [0,1,2])
fromList [1,2,3]
-- and so one ...
```

`reach'` calls `step` first on the set of states it is given, then on this list plus all the states that can be reached in one step, then again on all these plus the states reached in two steps, and so on ...

The function `reach`, which you have to complete, is designed to call `step [q]` only once for each reachable state, `q`.

Exercise 1

You should try to complete the following code, by replacing `undefined` with a suitable expression matching the template suggested: `foldr add (undefined) (undefined)`.

```
reach :: Ord q => (Set q -> Set q) -> Set q -> Set q
reach step qs =
  let add q qss
        | q `member` qss = qss
        | otherwise = undefined
         -- foldr add done todo
  in foldr add empty qs
```

Compare your implementation of `reach` with `reach'` . They should give the same answers.

Hint: To convey the idea of our use of `foldr`, we give the two undefined arguments suggestive names, **done** and **todo**. If `q` is already in `qss`, the call `add q qss` should return `qss`. However if `not(q `elem` qss)` then you should put `q` together with `qss` in place of **done** and put the states immediately reachable from `q` – that is, `step$ set[q]` in place of **todo**. So you should insert `q` in `qss`, to replace the first **done**; and then put `step$ set[q]` for the second **todo**. Then `foldr` will work through the todo list, ignoring states that are already done; as for the rest, it will add them one-by-one to the done list and process their adjacent nodes.

If you get stuck on this exercise, which is tricky, you can comment out the code for `reach`, temporarily, and rename `reach'` as `reach` to complete the rest of the tutorial. You can always come back later to implement the more efficient version.

Labelled transitions We represent labelled transitions in Haskell, using the following types:

```
type Sym = Char -- symbols
type Trans q = (q, Sym, q)
```

As a toy example, we take the numbers `[0..9]` as the states, and consider the following list of transitions:

```
toy_ts :: [Trans Int]
toy_ts = [ (n,intToDigit m, m*n `mod` 10) | n <- [0..9], m <- [3,7] ]
```

We use this to define a step function: `smallstep qs` $= \{(q,q') \mid \exists(q,a,q') \in$ `toy_ts` $.\ q \in$ `qs`$\}$.
This computes the states reachable in one step from `qs`.

Exercise 2

Translate the mathematical description of `smallstep` given above into Haskell by replacing the dots in the following skeleton

```
smallStep :: Set Int -> Set Int
smallStep qs = set[ ... | (q,_,q') <- toy_ts, ... qs ]
```

Use `reach'` to find which numbers are reachable from each of the numbers [0..9]:

Can you find any sets, `qs`, of states, such that `smallstep qs = qs`?
Can you find all such sets?

Draw a graph of the transition system, labelling each arrow with either 7 or 3. We will use it soon as a machine.

In this example, we took a set of transitions and produced a step function that we passed to `reach`. We can make this construction for any set of transitions.

Exercise 3

Use your answer to exercise 2 as a pattern, to define a function,

```
oneStep :: Ord q => [Trans q] -> Set q -> Set q
oneStep ts qs = set undefined
```

This should take a list `ts` of labelled transitions, then a list `qs` of states, and return the set of states that can be reached in one step from `qs` using transitions in `ts`.
For example, it should be the case that `smallstep qs = onestep toy_ts qs`.

0.1 Reachable states

We are now ready to look at reachability in the context of FSMs. We use the following Haskell type for FSM:

```
--   FSM states symbols transitions start finish
data FSM q = FSM (Set q) (Set Sym) [Trans q] (Set q) (Set q) deriving Show
mkFSM :: Ord q => [q] -> [Sym] -> [(q, Sym, q)] -> [q] -> [q] -> FSM q
mkFSM qs as ts ss fs = FSM qs' as' ts' ss' fs' where
  qs' = set qs
  as' = set as
  ts' = [ t | t@(q,a,q') <- ts, q`member`qs',q'`member`qs',a`member`as' ]
  ss' = set ss /\ qs'
  fs' = set fs /\ qs'
```

This is the same idea as in tutorial CL8, but here we make the use of sets explicit, and take more care. The function `mkFSM` makes lists into Sets, and also ensures that all components of the machine only reference states from the machine's set of states and symbols from its alphabet.

We will use our toy example to create a couple of FSMs, using the following definitions

```
g0 :: [Int] -> [Int] -> FSM Int -- q0 sets up the states, alphabet
g0 = mkFSM [0..9] "37" toy_ts   -- and transitions
eg0 = g0 [1]   [9]              -- we just have to add
eg1 = g0 [1,2] [9]              -- start and finish states
```

This gives us two FSMs that differ only in their start states.

To find the states reachable from a set `ss` of starting states, for a given set, `ts`, of transitions, we simply evaluate `reach (oneStep ts) ss`. (If you haven't yet implemented `reach` you can rename `reach'` and use that.)

Exercise 4

Define a function that computes the reachable states of an FSM.

```
reachableFSM :: Ord q => FSM q -> Set q
reachableFSM (FSM qs as ts ss fs) = undefined
```

Use this function to find which states are reachable for our two machines.

Do these machines recognise the same language, or two different languages?

Exercise 5

Write a function to prune the unreachable states from an FSM. The states of the pruned machine should be the reachable states of the original machine. You also have to eliminate transitions, and finish states, that mention the pruned states.

By definition, every start state is reachable, so these are unchanged; as is the alphabet.

As usual, you have to complete the `undefined` parts – and define the transitions and accepting states for the new machine.

```
pruneFSM :: Ord q => FSM q -> FSM q
pruneFSM fsm@(FSM qs as ts ss fs) =
  FSM qs' as ts' ss fs' where
    qs' = reachableFSM fsm
    ts' = undefined
    fs' = undefined
```

Your code should ensure that all components of the machine returned only reference states that belong to its set of states.

Hint: You should include only those transitions that start from a reachable state – if a transition starts from a reachable state then its destination state is reachable. Use `(/\)`, the set intersection function provided, to prune the finish states.

Test your function on the two machines, `eg0` and `eg1`; prune each of them.

- What can you say about the states of the pruned machines?
- Are the two pruned machines equivalent – do they recognise the same language?

We introduce a second property, closely related to reachability.

- A state `q` is **reachable** iff there is a path *from some start state* to `q`.
- A state `q` is **co-reachable** iff there is a path from `q` *to some finish state*.

Unless a state is co-reachable, it can never lead to an accepting state. So we can remove any states that are not co-reachable, without affecting the machine's behaviour.

Q. How do we identify the co-reachable states?

A. The co-reachable states are the reachable states of the reversed machine.

A state that is both reachable and co-reachable is a *live* state; the rest are *dead*.

Exercise 6

Complete the definition of reverseFSM.

```
reverseFSM :: Ord q => FSM q -> FSM q
reverseFSM (FSM qs as ts ss fs) = FSM qs as ts' ss' fs' where
    ts' = undefined
    ss' = undefined
    fs' = undefined
```

The code provided includes a `quickCheck` predicate, `prop_reverseFSM`, you can use to check `reverseFSM`, once you have implemented `ddelta` in Exercise 9.

Exercise 7

Define a function, `tidyFSM`, that leaves only states that are both reachable and co-reachable.

```
tidyFSM :: Ord q => FSM q -> FSM q
tidyFSM = undefined . undefined . undefined . undefined
```

Hint: your function should be a composition of `reverseFSM` and `pruneFSM` – using each of them twice. Test your function on our two examples.
Can you explain why the resulting machine is equivalent to the original?

NFA = FSA + ε

We add a list of pairs of states, `es :: [(q,q)]`, to represent machines that *may* include ε-transitions. (We say *may*, since the list of ε-transitions may be empty.)

```
data NFA q = NFA (Set q) (Set Sym) [Trans q] [(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 = NFA qs' as' ts' es' ss' fs' where
  qs' = set qs
  as' = set as
  ts' = [ t | t@(q,a,q')<-ts, q`member`qs',q'`member`qs',a`member`as']
  es' = [ e | e@(q,q') <- es, q`member`qs',q'`member`qs', q/=q']
  ss' = set ss /\ qs'
  fs' = set fs /\ qs'
```

We do not allow reflexive ε-transions; in addition to the checks we made before in `mkFSM`, `mkNFA` ensures that there are none.

In order to test our NFA and FSM implementations against each other, it is useful to be able to pass from FSM to NFA, and, for NFA with no ε-transions, to go back:

```
asNFA :: FSM q -> NFA q
asNFA (FSM qs as ts ss fs) = NFA qs as ts [] ss fs
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"
```

Here is an example of an NFA.

```
m3 :: NFA Int
m3 =  NFA [0..5] "ab" [(1,'a',2),(3,'b',4)] [(0,1),(2,3),(0,5),(4,5),(4,1)] [0] [5]
```

We introduced our operational model of FSM by considering how the *lit* states change in response to an input symbol. The key difference between FSM and NFA is that the collection of lit states for an NFA will always be ε-closed.

A set `qs` of states is **ε-closed** iff $\forall q \in$ `qs` . $q \xrightarrow{\varepsilon} q' \Rightarrow q' \in$ `qs` . This means that, any state reachable by a sequence of ε-transions from `qs` is already in `qs`.

Exercise 8

```
eStep   :: Ord q =>  [(q,q)] -> (Set q -> Set q)
eStep es qs = undefined
eClose :: Ord q => [(q,q)] -> (Set q -> Set q)
eClose es = reach (eStep es)
```

Complete the definition of `eStep`, to define a function, `eClose`.

Given a list of ε-transitions, `eClose` should return a function of type `Set q -> Set q` that computes the ε-closure of a set of states.

Test your code on the transitions in `m3`.

Converting an NFA to a DFA

You converted FSM to DFA in the FP9 tutorial. We will refactor that code to use our tools for reachability. Just as in FP9, you used the *powerset construction* to convert an FSM to an equivalent DFA, here you will use a powerset construction which follows the same pattern, adapted to take account of ε-transitions.

- The superstates of an FSM are sets of states of the original FSM.
  The superstates of an NFA are ε-closed sets of states of the original NFA.
- For the FSM conversion, the DFA makes a transition for symbol σ from superstate `superq` to the superstate `superq'` of states reached by one σ-labelled transition. For the NFA conversion the transition takes us to the ε-closure of this set.
- The accepting (super)states of the DFA are those which include some accepting state of the original NFA.
- The initial (super)state is the ε-closure of the set of start states of the original FSM.

We have already specified the superstates. The next step is to define the transitions.
First you will use `reach` to implement this step for the FSA-DFA case.

Exercise 9

Use `oneStep` to define the way the lit states of an FSM change for transitions `ts` and input symbol, σ.

```
ddelta :: (Ord q) =>  [Trans q] -> Sym -> (Set q -> Set q)
ddelta ts a = oneStep undefined
```

Hint: You need to follow any transition labelled with σ. The type of `oneStep` tells us that `undefined` should be replaced by an expression of type `[Trans q]`; use a list comprehension.

This function does the same job as the `ddelta` used in FP9, except that it takes a set of transitions as a parameter (rather than a machine with those transitions) and has a different order for its remaining parameters (FP9 used `ddelta fsm qs a`; here in CL9 we use `ddelta ts qs a`).

Acceptance for an FSM is defined by,

```
moveFSM :: Ord q => FSM q -> Sym -> FSM q
moveFSM (FSM qs as ts ss fs) x = FSM qs as ts (ddelta ts x ss) fs

acceptsFSM :: (Ord q) => FSM q -> String -> Bool
acceptsFSM (FSM _ _ _ ss fs) "" = (not . Set.null)(ss /\ fs)
acceptsFSM fsm (x : xs) = acceptsFSM (moveFSM fsm x) xs
```

For the NFA to DFA conversion we must combine `eClose` and `ddelta`. We can make a chain of ε-transitions before and after every labelled transition.

Exercise 10

Use `ddelta` and `eClose` to define the way the lit states of an NFA change for transitions `ts` and input symbol, σ.

```
eddelta :: (Ord q) => [Trans q] -> [(q,q)] -> Sym -> Set q -> Set q
eddelta ts es a = undefined . undefined . undefined
```

Hint: The machine can move by ε-transitions, followed by an a-transition, followd by ε-transitions. The operation on *superstates* should thus be a composition of functions given by applying eClose and ddelta to appropriate arguments.

Using eddelta we can define moveNFA and acceptsNFA to match the functions you wrote in CL8 for FSMs.

```
moveNFA :: Ord q => NFA q -> Sym -> NFA q
moveNFA (NFA qs as ts es ss fs) a = NFA qs as ts es (eddelta ts es a ss) fs


acceptsNFA :: (Ord q) => NFA q -> String -> Bool
acceptsNFA (NFA _ _ _ es ss fs) "" = (not . Set.null)(eClose es ss /\ fs)
acceptsNFA nfa (x : xs) = acceptsNFA (moveNFA nfa x) xs
```

The next step in the subset construction is to identify the start superstate as the ε-closure of the set of start states; then compute the superstates reachable from the starting superstate. We will use reach to do this. We need a step function to say which superstates are reachable in one step from a set of superstates: a function of type Set(Set q) -> Set(Set q).

In FP9 you defined next :: FSM q -> [[q]] -> [[q]]. We will define a corresponding superstep function. Again our function does the same job but has a different type.

```
next :: Ord q =>  [Trans q] -> (Set Sym)-> Set(Set q) -> Set(Set q)
next ts as qqs = set [ddelta ts a qs | qs <- toList qqs, a <- toList as ]
```

The step function required by reach is given by next ts as, and we define the reachable superstates by, qs' = reach (sstep as ts) [ss] . We use ddelta, to define the transitions of the DFA, like this : ts' = [(qs, a, ddelta ts a qs)| qs <- superqs, a <- as] .

Putting this all together we define the DFA corresponding to a given FSM

```
fsm2dfa :: Ord q => FSM q -> FSM (Set q)
fsm2dfa (FSM qs as ts ss fs) =
  FSM qs' as ts' ss' fs' where
  qs' = reach (next ts as) ss'
  ts' = [ (qs, a, ddelta ts a qs) | qs <- toList qs', a <- toList as ]
  ss' = set [ss]
  fs' = set [ qs | qs <- toList qs', (not . Set.null)(qs/\fs) ]
```

Exercise 11

For an NFA we will use eddelta in place of ddelta.
You should define a function eNext to use in place of next

```
eNext :: Ord q =>  [Trans q] -> [(q,q)] -> (Set Sym)-> Set(Set q) -> Set(Set q)
eNext ts es as qss = set undefined
```

Hint: Use the pattern we used for next, but with eddelta.

Exercise 12

Now you can write the function that converts an NFA to a DFA – represented as an NFA with an empty list of ε-transitions, a single start state and exactly one transition (q,a,q') for each q,a pair .

```
nfa2dfa :: Ord q => NFA q -> NFA [q]
nfa2dfa (NFA qs as ts es ss fs) =
  FSM qss as ts' es' ss' fs' where
  qss = undefined
```

```
        ts' = undefined
        es' = undefined
        ss' = undefined
        fs' = undefined
```

Hint: don't forget to `eClose` the set of starting states!

Exercise 13

Write a function `intNFA :: NFA a -> NFA Int` that translates an `NFA q` (whether deterministic or not) into an equivalent `NFA Int` which has a state space `[0..n-1]`, where `n` is the number of states in the original NFA.

```
intNFA :: (Ord q, Show q) => NFA q -> NFA Int
```

Hint: look at the code from FP9

Exercise 14

Write a function a reverse function for NFA.

```
reverseNFA :: Ord q => NFA q -> NFA q
```

The function `tidyNFA` produces a minimal DFA for a given NFA.

```
tidyNFA :: Ord q => NFA q -> NFA Int
tidyNFA =  intNFA . nfa2dfa . reverseNFA . nfa2dfa . reverseNFA
```

regex → NFA

In this section you will implement functions that allow you to create an FSM for any regex. There are some QuickCheck properties already defined to help you along. 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 Thompson's construction[4] to build an NFA that recognises the language defined by a given regular expression.

The key idea of this 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.

```
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
```

We can convert any FSM to an NFA, trivially, by giving it an empty list of ε-transitions. We can convert any NFA to Thomson form by adding two new states and some ε-transitions. 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)
thompson :: Ord q => NFA q -> NFA (QF q)
thompson (NFA qs as ts es ss fs) =
  mkNFA qs' as ts' es' ss' fs'
  where qs' = Q: F : map E qs
        ts'  = mapTrans E ts
```

[4]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.

```
es'  = [ (E q, E q') | (q,q') <- es] ++
       [ (Q,E q) | q <- ss ] ++ [ (E q,F) | q <- fs ]
ss'  = [Q]
fs'  = [F]
```

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

Exercise 15

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

(b) Define a value `nullNFA :: NFA ()` that represents a Thompson NFA that accepts the empty language ∅, with no strings.

(c) Define a value `dotNFA :: NFA Bool` that represents a Thompson NFA that accepts every single-character string from our alphabet – the language `[[sym] | sym <- ['a'..'z']]`.

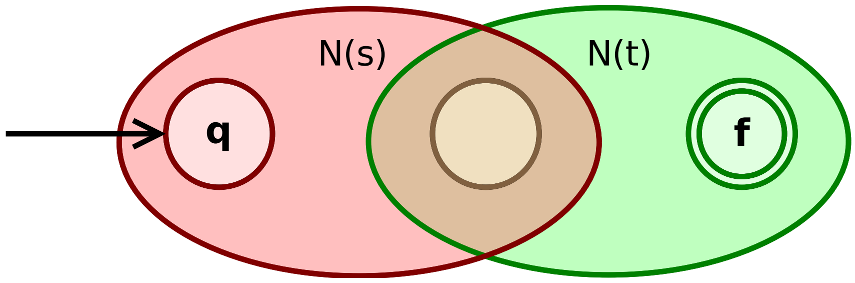
Exercise 16

The concatenation `N(st)` of two Thompson NFAs `N(s)` and `N(t)` is an automaton with the union of the two alphabets that accepts an input word if `N(s)` accepts some prefix of the input word and `B` accepts the rest of the word.

Complete the following definition by replacing every `undefined`. The idea is to match the diagram so that we use the final state of the Left machine to connect the two. We remove the start state of the Right machine, and we have to replace every transition from it by a transition from the final state of the Left machine.

```
tconcatNFA :: (Ord a, Ord b) => NFA a -> NFA b -> NFA (Either a b)
tconcatNFA (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' \\ ss')
    as'' = as \/ as'
    ts'' = mapTrans Left ts
           ++ [ undefined | (q,a,q') <- ts', q == the ss']
           ++ [ undefined | (q,a,q') <- ts', q /= the ss' ]
    es'' = [ (Left q, Left q') | (q,q') <- es ]
           ++ [ undefined | (q,q') <- es', q == the ss']
           ++ [ undefined | (q,q') <- es', q /= the ss' ]
    ss'' = mapS Left ss
    fs'' = mapS Right fs'
    the xs = (\[x] -> x) (toList xs) -- thompson
```

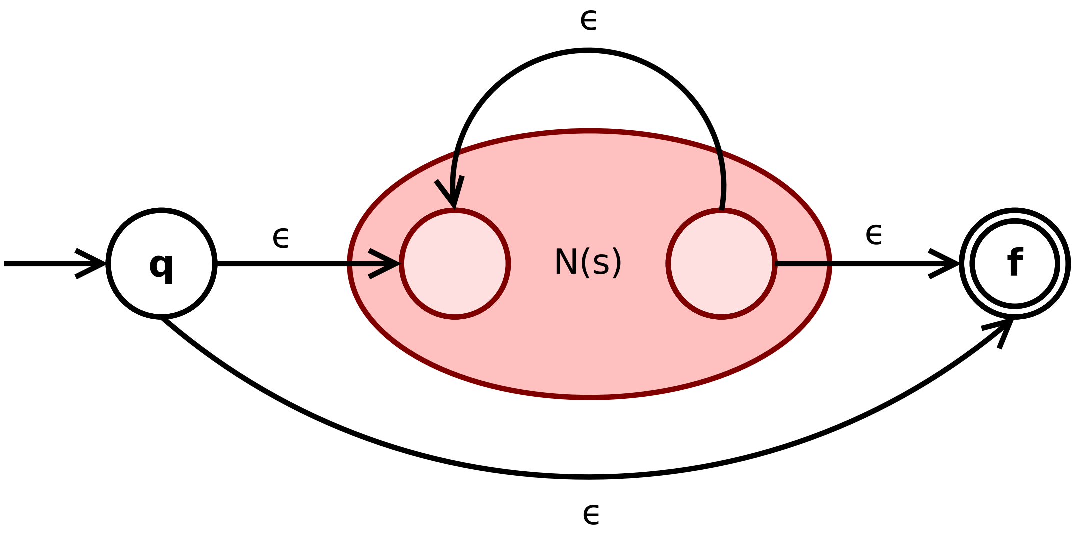
Provided its arguments are Thompson it should return a Thompson NFA that recognises the concatenation of the languages of its two arguments.



If you have problems with this exercise, you can always simply apply the function `thompson` to the generic `concatNFA` provided, but your machines won't be as compact.

Exercise 17

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.



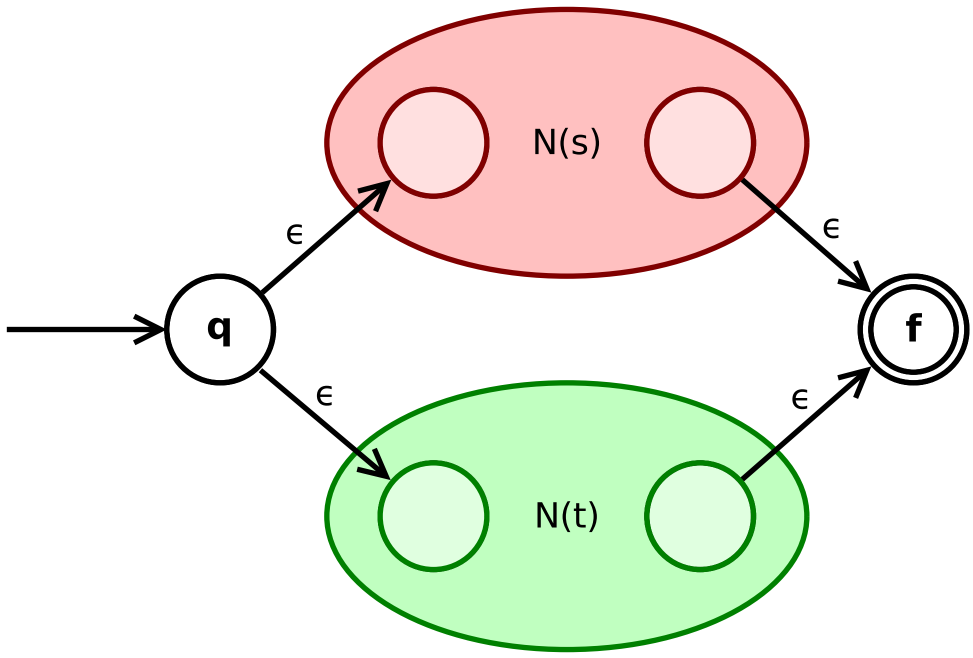
(a) We use the type `QF q` introduced above. Replace the `undefined`s to complete the function so that it returns the Kleene star closure of the input automaton. Your function should work on any Thompson NFA.

```
starNFA :: Ord q => NFA q -> NFA (QF q)
starNFA (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' = (Q,F) : [(E f, E q) | f <- toList fs, q <- toList ss ]
              ++ [ undefined | (q,q') <- es]
              ++ [ undefined | q <- toList ss ]
              ++ [ undefined | q <- toList fs ]
        ss' = set [Q]
        fs' = set [F]
```

(b) Try out your code by writing some regular expressions of your own and testing them on sample strings.

Exercise 18

The union of two NFAs `N(s)` and `N(t)` is an NFA `N(s|t)` which accepts a word if either `N(s)` or `N(t)` accepts it.



```
unionNFA ::
tunionNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (QF (Either q q'))
```

You should construct the union by placing the two machines side-by-side, using the example given by `unionFSM`. The type signature is a giant hint. You should use the Haskell type `Either q q'` for the states of a machine that includes a copy of each of the two arguments. Conceptually, we will run them in parallel, to give an `NFA (Either q q')` that recognises the union of the languages, and then use the function `thompson` to convert it to a Thompson NFA.

Exercise 19

In the last part of this tutorial, you will implement two additional operations, complement and intersection. In order to work with the regular operations defined for the Thompson construction, you should return a Thompson NFA for all but the complete NFA (a complete NFA cannot be Thompson).

(a) Recall that a machine is complete if for each state, symbol pair, q, a there is at least one state q' with a transtion (q, a, q').

Write a function `completeNFA :: (Ord q) => NFA q -> NFA (Maybe q)` that returns a complete NFA equivalent to the original.

It is then trivial to write

```
complementNFA :: Ord q => NFA q -> NFA (Maybe q)
complementNFA nfa = let NFA qs as ts es ss fs = completeNFA nfa
                    in  NFA qs as ts es ss (qs \\ fs)
```

(b) The intersection of two NFAs `A` and `B` is an NFA which accepts a word if and only if both `A` and `B` accept it.

You will use the product construction to implement the intersection. For FSM we defined the `productFSM` as follows:

```
productFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (q,q')
productFSM (FSM qs as ts ss fs) (FSM qs' as' ts' ss' fs')
  = FSM qs'' as'' ts'' ss'' fs'' where
    qs'' = Set.cartesianProduct qs qs'
    as'' = (as \/ as')
    ts'' = [((x,y), a, (x',y')) | (x,a,x') <- ts, (y,a',y') <- ts', a == a']
    ss'' = Set.cartesianProduct ss ss'
    fs'' = Set.cartesianProduct fs fs'
```

The code for `productNFA` will be similar in many respscts, but you also need to think about ε-transitions.

```
productNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (q,q')
productNFA (NFA qs as ts es ss fs) (NFA qs' as' ts' es' ss' fs')
  = NFA qs'' as'' ts'' es'' ss'' fs'' where
  qs'' = undefined
  as'' = undefined
  ts'' = undefined
  es'' = undefined
  ss'' = undefined
  fs'' = undefined

intersectNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (q,q')
intersectNFA = productNFA
```

Hint: The product machine may make an ε-transition `((x,y),(x,y'))` if the Right machine has an ε-transition `(y,y')`; similarly the product machine may make an ε-transition `((x,y),(x',y))` if the Left machine has an ε-transition `(x,x')`; and of course there should be an ε-transition `((x,y),(x',y'))` if the left has `(x,x')` and the Right has `(y,y')`.

(c) Does your product machine produce a Thompson machine if its arguments are both Thompson?

(d) Use the QuickCheck properties at the bottom of tutorial9.hs to write your own tests.

Exercise 20

This exercise is optional.

The following data type represents regex extended with Boolean operations:

```
data Regex =
  S String | Regex0 | Regex1
  | (:|:) Regex Regex -- alternation
  | (:&:) Regex Regex -- intersection
  | (:>:) Regex Regex -- concatenation
  | Not  Regex        -- negation
  | Star Regex        -- Kleene-star

regex2nfa :: Regex -> NFA Int
```

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

Write a function that first produces a minimal DFA (represented as an `FSM Int`) for that regex, then applies `tidyFSM` to any black-hole state.

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?