FSM(ii)

In this tutorial you wil 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 ordered lists to represent sets of states, with the type abbreviation, `type Set q = [q]` .

The *set invariant* is that the list should be a strictly increasing list – that is its elements should be in increasing order, with no duplicates. We will use the type abbreviation in signatures to indicate that a parameter must satisfy the invariant, or that the value returned will satisfy it. We have not used a module to make these sets abstract, so we must take care to ensure that the invariant is maintained in our code.

We use the following operations on sets (implementations are given at the end of this document).

```
set :: Ord a => [a] -> Set a
insert :: Ord a => a -> Set a -> Set a
(\/) :: Ord a => Set a -> Set a -> Set a -- union
(/\) :: Ord a => Set a -> Set a -> Set a -- intersection
(\\) :: Ord a => Set a -> Set a -> Set a -- setminus
elem :: Ord a => a -> Set a -> Bool      -- membership
```

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[1] crops up in many areas of informatics.

We will define a function

```
reach :: Ord p => ([p] -> [p]) -> [p] -> [p]
```

The first argument of `reach` is a function `step :: [q] -> [q]` such that `step ss` returns the list 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 :: [p] -> [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 p => ([p] -> Set p) -> Set p -> Set p
reach' step gs =
  if gs' == gs then gs else reach' step gs'
  where gs' = gs \/ step gs
```

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

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`. `reach'` calls `step` first on the list 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` is designed to call `step [q]` only once for each reachable state, `q`.

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

Exercise 1

You should try to complete the following code, by replacing each `undefined` with a suitable expression.

```
reach :: Ord q => ([q] -> [q]) -> [q] -> Set q
reach step qs =
  let add q qss
        | q `elem` qss = qss
        | otherwise =
          foldr add undefined undefined
  in foldr add [] qs
```

The idea is that if `not(q `elem` qss)` then we should insert `q` in `qss`, to replace the first undefined; and then put `step [q]` for the second undefined so that `foldr` can check these, one-by-one, and add them and their neighbours only if they have not yet been processed.

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

If you get stuck on this exercise, which is tricky, you can use `reach'` to complete the rest of the tutorial, and come back later to implement `reach`.

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

```
type Sym = Char
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 = [ (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 :: [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]:

```
map ( reach' smallstep . (\x -> [x]) ) [0..9]
```

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

Draw a picture 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,

```
one_step :: Ord q => [Trans q] -> [q] -> Set q
one_step 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`.

We are now ready to look at reachability in the context of FSMs.

0.1 Reachable states

We use the following Haskell type for FSM:

```
--   FSM states symbols transitions start finish
data FSM q = FSM (Set q) [Sym] [Trans q] (Set q) (Set q) deriving Show
mkFSM qs as ts ss fs = FSM (set qs) as ts (set ss) (set fs)
```

This is the same as in tutorial CL8, but here we make the use of sets explicit. The function `mkFSM` establishes the set invariant – by using the function `set = nub.sort` to put each set in canonical form. Our code includes a predicate that checks the invariant in our `quickCheck` properties.

```
prop_sorted xs = and[ x < y | (x,y) <- zip xs (tail xs) ]
prop_invariantFSM (FSM qs as ts ss fs) = -- g0 is incomplete ...
  prop_sorted qs && prop_sorted ss && prop_sorted fs
```

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

```
g0 :: [Int] -> [Int] -> FSM Int
g0 = mkFSM [0..9] "37" [(n,intToDigit m, m*n `mod` 10)| n <- [0..9], m <- [3,7]]
eg0 = g0 [1]   [9]
eg1 = g0 [1,2] [9]
```

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 (one_step ts) ss`. (If you don't yet have `reach` you can use `reach'`.)

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) = reach (one_step ts) ss
```

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 reachable states of the original machine, `qs'`, are the states of the pruned machine. 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
```

Hint: You should include only those transitions that start from a reachable state – by definition, 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?

- A state `q` is *reachable* iff there is 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 accpeting state. So we could also 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.

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
```

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 (the list may be empty) include $\boldsymbol{\varepsilon}$-transitions.

```
data NFA q = NFA (Set q) [Sym] [Trans q] [(q,q)] (Set q)(Set q) deriving Show
mkNFA qs as ts es ss fs =
  NFA (set qs) as ts [ e | e@(q,q') <- es, q/=q'] (set ss) (set fs)
```

We do not allow reflexive $\boldsymbol{\varepsilon}$-transions; `mkNFA` ensures that there are none.

We will use an example NFA from FP9.

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

Exercise 8

Define a function `eClose` to compute the ε-closure of a list of states, given a list of ε-transitions.

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

Test your code on the transitions in `m3`.

0.2 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. At each stage you will write the corresponding code for NFA to DFA conversion.

Just as in FP9, you used the "powerset construction," to convert an FSM to an equivalent DFA, you will use a poserset construction with the same steps, 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 specified the superstates. The next step is to define the transitions. First we use our reachability tools to present this step for the FSA-DFA case.

Exercise 9

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

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

Hint: You need to use only the transitions labelled with σ. The type of `one_step` 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 list 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`).

For The NFA to DFA conversion we first use `ddelta` then `eClose`.

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) => ts -> es -> Sym -> [q] -> Set q
eddelta ts a = undefined . undefined
```

Hint: The *target superstate* is the ε-closure of the set of states to which the machine can move, starting from one of the source states, given the input symbol.

The next step is to identify the superstates reachable from the start state. We will use `reach` to do this. We need a step function to say which superstates are reachable in one step from a list of superstates: a function of type `[[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] -> [Sym] -> [[q]] -> Set (Set q)
next ts as qss = set [ddelta ts a qs | qs <- qss, a <- 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 [q]
fsm2dfa (FSM qs as ts ss fs) =
  FSM qss as ts' ss' fs' where
  qss = reach (next ts as) [ss]
  ts' = [ (qs, a, ddelta ts a qs) | qs <- qss, a <- as ]
  ss' = [ss]
  fs' = [ qs | qs <- qss, or[ q`elem`fs | q <- qs ]]
```

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)] -> [Sym] -> [[q]] -> Set (Set q)
eNext ts es as qqs = set undefined
```

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

Exercise 12

Now you can write the function that converts from NFA to DFA.

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

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

Exercise 13

If you have not done this already in FP9, write a function `toIntNFA :: NFA a -> NFA Int` that translates a `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.

```
intFSM :: (Ord q, Show q) => FSM q -> FSM Int
```

1 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 constructionto 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 transtitions ending in its start state, and no transitions starting from its accepting state. We call sucha machine a Thompson NFA.

```
isThompson (NFA qs as ts es [s] [f]) =
  null [ q | (q,_,q') <- ts, s==q'||f==q] && null [ q | (q,q') <- es, s==q'||f==q ]
isThompson _ = 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
        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 14

(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 no strings.

(c) Define a value `fullNFA :: NFA ()` that represents a Thompson NFA that accepts every string on the alphabet `['a'..'z']`.

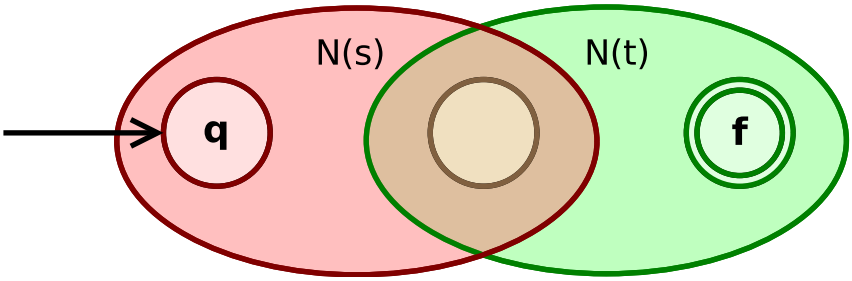
Exercise 15

The concatenation `N(st)` of two Thompson NFAs `N(s)` and `N(t)` 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.

Implement a function

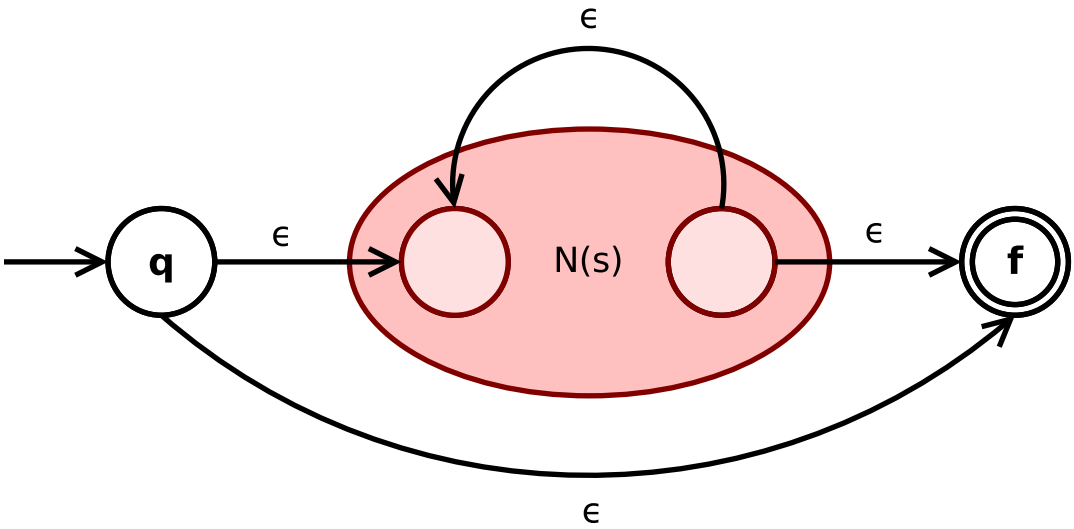
```
concatNFA :: (Ord q, Ord q') => NFA q -> NFA q' -> NFA (Either q q')
```

which returns the Thompson concatenation of the input Thompson NFAs.



Exercise 16

The Kleene star closure of a 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.

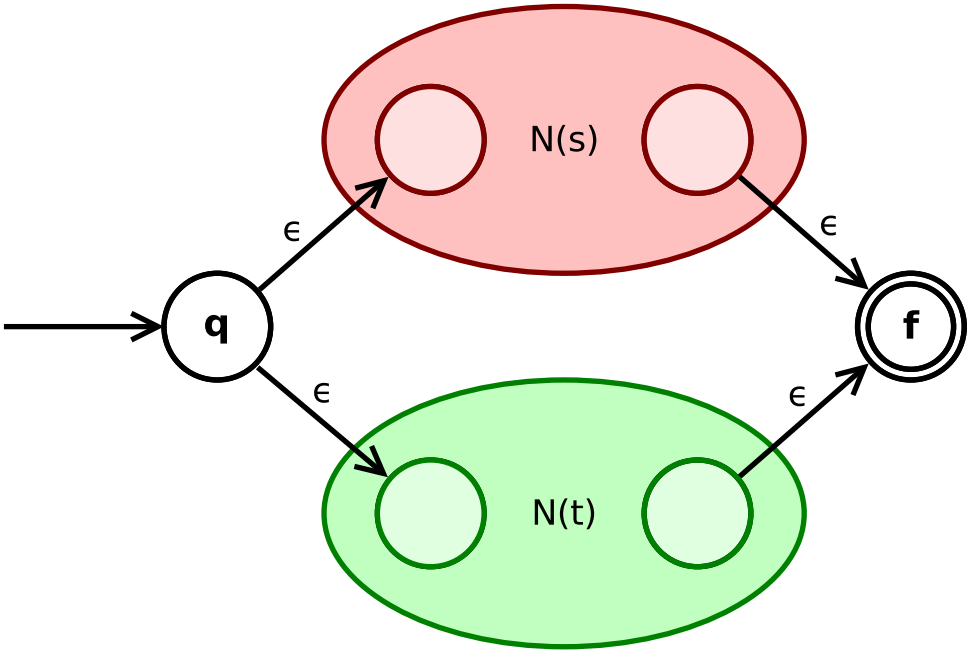


(a) Use the type `QF q` introduced above to write a function `kstar :: NFA q -> NFA(QF q)` that returns the Kleene star closure of the input automaton. Your function should work on any Thompson NFA.

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

Exercise 17

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 :: (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. The type signature is a giant hint. You should use the Haskell type `Either q q'` to place the two machines in parallel. Conceptually, we will run them in parallel, and then convert to a Thompson NFA.

Exercise 18

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

(b) Write a function `complementNFA :: (Ord q) => NFA q -> NFA (Maybe q)` that returns an NFA which accepts an input if and only if the input NFA rejects it. Use the provided QuickCheck property to test your function.

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

Implement a function

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

which returns the intersection of the input NFAs. Your function should work on any NFA (whether deterministic or not).

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

2 Sets as ordered lists

```
type Set a = [a]
set :: Ord a => [a] -> [a]
set = nub.sort

insert :: Ord a => a -> Set a -> Set a
insert x ys@(y:yt) =
    case compare x y of
     LT -> x : ys
     EQ -> ys
     GT -> y : insert x yt
insert x [] = [x]

(\/) :: Ord a => Set a -> Set a -> Set a
xs@(x:xt) \/ ys@(y:yt) = -- ⋃ union
  case compare x y of
   LT -> x : (xt \/ ys)
   EQ -> x : (xt \/ yt)
   GT -> y : (xs \/ yt)
[] \/ ys = ys
xs \/ [] = xs


(/\) :: Ord a => Set a -> Set a -> Set a
xs@(x:xt) /\ ys@(y:yt) = -- ⋂ intersection
  case compare x y of
   LT -> (xt /\ ys)
   EQ -> x : (xt /\ yt)
   GT -> (xs /\ yt)
_ /\ _ = []

(\\) :: Ord a => Set a -> Set a -> Set a
xs@(x:xt) \\ ys@(y:yt) =
  case compare x y of
   LT -> x: (xt \\ ys)
   EQ -> (xt \\ yt)
   GT -> (xs \\ yt)
[] \\ _  = []
xs \\ [] = xs

elem :: Ord a => a -> Set a -> Bool
a`elem`[] = False
a`elem`xs@(x:xt) =
  case compare a x of
   LT -> False
   EQ -> True
   GT -> a`elem`xt

delete :: Ord a => a -> Set a -> Set a
delete x ys@(y:yt) =
  case compare x y of
   LT -> ys
   EQ -> yt
   GT -> y : delete x yt

union :: Ord a => [Set a] -> Set a
union = foldr (\/) []

unionMap :: (Ord q, Foldable t) => (a -> Set q) -> t a -> Set q
unionMap f = foldr ((\/).f) []

lookup :: Ord a => a -> [(a,b)] -> b
lookup x (t@(a,b) : tt) =
   case compare x a of
    LT -> lookup x tt
    EQ -> b
    GT -> error "lookup: nothing found"

prop_sorted xs = and [ x < y | (x,y) <- zip xs (tail xs) ]
```