FSM (i)

Objectives: In the videos, you've learned about finite state machines (FSMs), also known as finite state automata (FSA), both deterministic (DFA) and nondeterministic (NFA), and you've seen how the latter can be transformed into the former.

This tutorial lets you play with finite state machines, using the FSM workbench, implement finite state machines in Haskell, and transform an arbitrary FSM to a DFA.

Tasks: Exercises 1, 2, and 7 *do not require submission.*
Exercises 3, 4a, 4b, 6, 8, 10, and 11 are *mandatory.* Exercises 4c, 5, 9, and 12–15 are *optional.*

Submission: Submit a file cl-tutorial-9 (image or pdf) with your answers that do not require programming, and the file Tutorial9.hs with your Haskell code.

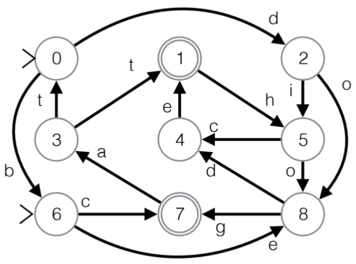
Deadline: The deadline for this tutorial is Saturday, 21st of November, 4PM UK time.

1 FSM

We begin with a couple of definitions.

A **finite state machine**[1] has five components, $(Q, \Sigma, \Delta, S, F)$:

Q a finite set of states
Σ a finite alphabet of symbols
$\Delta \subseteq Q \times \Sigma \times Q$ a set of transitions
$S \subseteq Q$ starting states
$F \subseteq Q$ accepting states



We draw an FSM as a directed graph whose nodes are the states and whose arrows are transitions. This example has nine states, an alphabet with nine symbols, `"aeiocdght"` , fifteen transitions, two start states, and two final states. We call this machine `eg1` .

A word is accepted by the machine if there is a path following a chain of transitions, from a start state to a finish state, such that the labels on the transitions spell out the word.
For example, the word "dog" is accepted by the path $0 \to^d 2 \to^o 8 \to^g 7$.

1. *No submission.*
   (a) How many English words can you find that are accepted by this machine?
   (b) Can you make a machine that accepts only these words? (Try this, but don't worry if you don't succeed – you will soon have the tools to do this.)
   (c) Design your own FSM with at most nine states, as many start-states, finish-states, and transitions as you like, and as many lower-case characters as you like as alphabet. Your machine should accept as many English words as you can manage, but no non-words.

   Use the words file on your machine, usually stored in `/usr/share/dict/words` or `/usr/dict/words` as an oracle if you want to make a determined attempt at this question – but that is definitely optional.

DFA We say a machine $M = (Q, \Sigma, \Delta, S, F)$ is *deterministic* iff

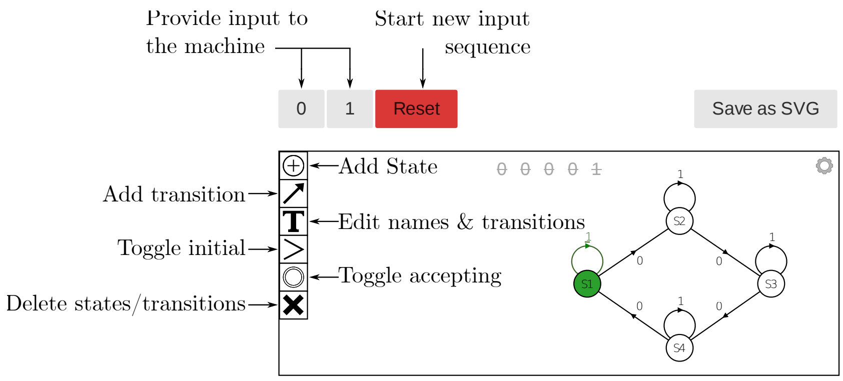
- M has *exactly* one start state $S = \{q_0\}$ and
- Δ represents a total function $Q \times \Sigma \to Q$ which means that for each state,symbol pair, $(q, \sigma) \in Q \times \Sigma$, there is exactly one q' such that $(q, \sigma, q') \in \Delta$.

[1] Many treatments allow only a single start state.
However, the theory is much smoother, and the coding is simpler, if we allow an arbitrary set of start states.

2 The FSM workbench

The FSM workbench, designed and implemented by Matthew Hepburn, is an interactive tool for designing and simulating machines. You may find it useful for testing your ideas. The workbench provides tools for creating, editing, and simulating finite state machines. The illustration shows the function of each tool.

You can toggle each tool on/off by clicking it. When no tools are active you can drag the states of your FSM to rearrange the layout.



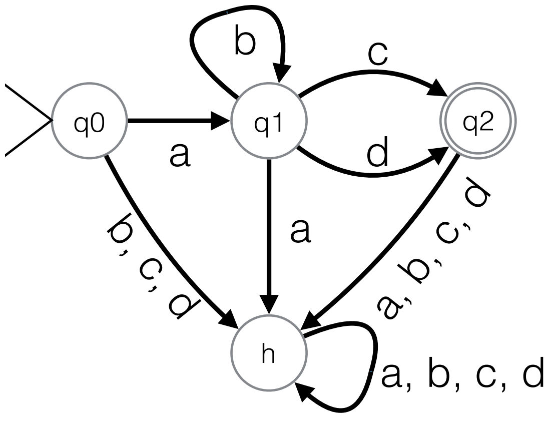
2. *No submission.* Work through the first half of the exercises on the workbench – click on **Exercises** to start – ending when you reach the introduction to ϵ-transitions..

The workbench uses a graphical representation of FSMs. Nodes (circles) represent states. Accepting states are marked with an inner circle. The initial start state is indicated with an arrowhead.

Initial
Accepting

A change from one state to another is called a *transition*. Edges (arrows) represent transitions; they are labelled with symbols from the alphabet.

Here is a simple DFA with four states, only one of which is accepting.



A simple DFA

Given a sequence of input symbols, we can simulate the action of the machine. Place a token on the start state, and taking each input symbol in turn, move it along the arrow labelled with that symbol. In a DFA there is exactly one such arrow from each state.

3. Which of the following strings are accepted by this machine?

   (a) `"abbd"` (b) `"ad"` (c) `"aab"` (d) `"abbbc"` (e) `"ac"`

* * * * * *

To describe the strings accepted by this DFA we write `ab*(c|d)` to mean that it accepts `a` (an `'a'`), followed by `b*` (any number (including zero) of `'b'`s), followed by `c | d` (`'c'` or `'d'`).
This is an example of a *regular expression* (regex).

We may apply the operations `*` `|` and juxtaposition to any regular expressions; `*` takes precedence over juxtaposition, which takes precedence over `|` so, for example, `a|cb*|(c|d)*` accepts any string that is either an `a`, or a followed by any number (including zero) of `b`s, or any string of `c`s and `d`s (including the empty string).

The black-hole convention In the simple DFA example above, the state `h` is *not an accepting state*; it has the property that,
if we ever get to `h` *we can never escape.*

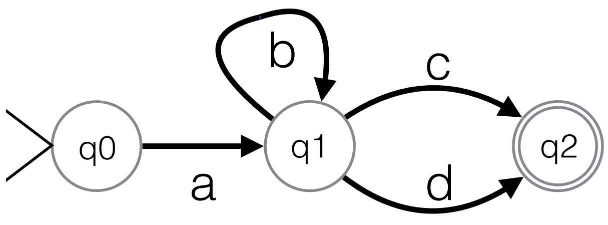
We call such a state a *black hole* state.

We may draw a simpler diagram for the machine if we omit the black hole state. It is easier to see the structure without it.

In a DFA, there is a transition from each state, for each symbol of the alphabet. We can recreate the missing transitions using the

black hole convention:

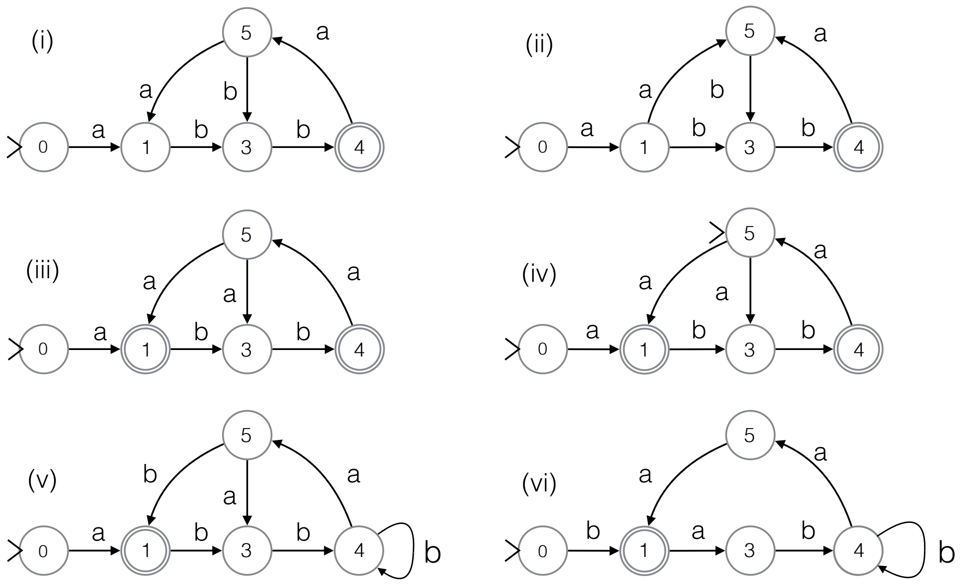
> *any missing transitions take us to a black hole state, which is not drawn*



A very simple DFA

With the black hole convention we can represent a DFA with a diagram in which, for each symbol of the alphabet, there is *at most one* transition from each state.

4. Consider the six FSMs in the diagram, answer the questions given below.



   (a) Which of these FSM are DFA, assuming the black-hole convention?

   (b) Which of the following strings are accepted by the first FSM (i)?

      i. `abb`
      ii. `abbaa`
      iii. `ab`
      iv. `aabbab`
      v. `babaaab`
      vi. `aabb`
      vii. `abbbbab`
      viii. `abbaabb`

   (c) *Optional* Choose *any one* of these machines and **try**, but don't try too hard,[2] to give a regular expression that matches the strings accepted by the machine.

5. *Optional* The product of two DFA, described in Chapter 30 of the textbook, accepts the intersection of the languages accepted by each machine.

   (a) Use **pen and paper**, produce the product of the machines, **M** and **N**, in Figure 1.

   (b) Is the product of two DFA a DFA? Explain your answer.

[2]**Try** means we don't expect you to succeed with all the examples – tools for tackling the harder ones will be covered in a future tutorial.

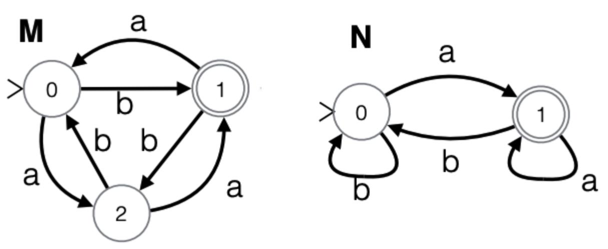


Figure 1: DFAs for question 5

3 FSM in Haskell

We will begin with FSMs without epsilon transitions; we'll introduce these next week.

An FSM is constructed from five components corresponding to the formal definition:

```
 --  FSM           states  symbols   transitions    starting accepting
 --                  Q      Sigma       delta          S         F
 --                  qs       as         ts            ss        fs
 data FSM q = FSM (Set q) (Set Sym) (Set (Trans q)) (Set q)   (Set q) deriving Show
```

Here, the type variable `q` represents the type of states; `Sym` is a synonym for `Char`, and `Trans q` is the type of labelled transitions, defined by:

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

We use the `Data.Set` library to represent the sets used in the formal definition (because the library uses ordered trees to represent sets, most of our functions will require `(Ord q)`). The transition function, Δ, is represented by the set, δ, of labelled transitions. $\delta = \{(q, s, q') \mid q' \in \Delta(q, s)\}$.

In the code for this tutorial we've added infix operators `/\` and `\/` for intersection and union of sets. You can use `toList :: Ord q => Set q -> [q]` and `fromList :: Ord q => [q] -> Set q` to convert sets to lists and vice-versa. You will have to use these if you want to use list comprehensions for some exercises. As an example consider the following definition.

```
 cartesianProduct :: Set x -> Set y -> Set (x,y)
 cartesianProduct a b = fromList [ (a, b) | a <- toList as, b <- toList bs ]
```

We will need this function next week, it's actually already defined in `Data.Set` .

Working with sets is a change from working with lists, but many of the ideas you're familiar with for lists in Haskell will carry over to sets. In particular, the functions `filterS` and `mapS` that we've imported for you from `Data.Set` :

```
 filterS :: Ord a => (a -> Bool) ->  Set a -> Set a
 mapS :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
```

work for sets, pretty much as the `Prelude` versions of `filter` , and `map` do for lists.

We provide a convenience function to construct an FSM from five lists, which we convert to sets to form the five components of the FSM:

```
 mkFSM :: Ord q => [q] -> [Sym] -> [Trans q] -> [q] -> [q] -> FSM q
 mkFSM qs as ts ss fs =  -- construct an FSM from five lists
   FSM (fromList qs) (fromList as) (fromList ts) (fromList ss) (fromList fs)
```

We will normally use the following variables for each component of the machine:

`qs :: Set q` the states, Q;

`ss :: Set q` the *starting* states, S;

`as :: Set Sym` the *alphabet* of symbols, Σ;

`fs :: Set q` the *final*, or accepting, states, F.

`ts :: Set (q,Sym,q)` the *transitions*, δ;

Our first example from Section 1 may be encoded as follows:

```
eg1 = mkFSM [0..8] "abcdeghiot"
  [(0,'d',2),(0,'b',6),(1,'h',5),(2,'i',5),(2,'o',8),(3,'t',0),(3,'t',1),(4,'e',1)
  ,(5,'c',4),(5,'o',8),(6,'c',7),(6,'e',8),(7,'a',3),(8,'d',4),(8,'g',7)]
  [0,6] [1,7]
```

We prefer to encode this (equivalently, but) slightly differently:

```
eg1 = mkFSM [0..8] as
  [(0,d,2),(0,b,6),(1,h,5),(2,i,5),(2,o,8),(3,t,0),(3,t,1),(4,e,1)
  ,(5,c,4),(5,o,8),(6,c,7),(6,e,8),(7,a,3),(8,d,4),(8,g,7)]
  [0,6] [1,7]
  where as@[a,b,c,d,e,g,h,i,o,t]="abcdeghiot"
```

This uses an *as-pattern*. The @ pattern lets you give a name to a variable while also requiring it to match a given pattern – it also allows you to name the components of that pattern. We will use @-patterns again later . . .

Instructions: Your main goal for the remainder of this tutorial is to produce a Haskell function that implements the powerset construction that converts any FSM to a DFA.

We have provided a code skeleton for this – but it is sprinkled with `undefined` expressions for which you must find appropriate replacements.

You will have to work with `Set` s instead of `List` s, but many of the functions you're familiar with for lists have counterparts for sets; for example, `filterS` and `mapS` which we've mentioned already.

Furthermore, lots of other functions you're familiar with, including, `or` , `and` , `any` , `all` , work with sets, just they do for lists.

The only time you need to use `toList` and `fromList` , in the code you'll write below, is when you want to use a list comprehension – and when that's needed we'll spell it out for you. In this and the following questions, we give you a template for the solution, with some `undefined` terms. Your task is to replace each occurrence of `undefined` with appropriate code.

Here's a first example of this style of question; your task is to replace two occurrences of `undefined` .

6. Complete the function `isDFA` so that it checks whether an FSM is a DFA.

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

3.1 Running a machine

Now we introduce Haskell functions that define how an FSM runs. We first picture an FSM as a machine with flashing lights – each state is a light; the start states are lit (on); the rest are off. The machine has a keyboard, with a key for each symbol $\sigma \in \Sigma$. Pressing a key, σ, (we may only press one at a time) changes the lights. We call the states whose light is on the *active states*. A state, q', is active after the key-press iff there is an arrow $q \to^{\sigma} q'$, from some state q that was active before, to q', labelled with σ . The next set of active states is given by:

$$\{q' \mid \exists q \in S \,.\, (q, \sigma, q') \in \Delta\}$$

The function `transition` translates this mathematical definition into Haskell to define how the lights change.

```
transition :: (Ord q) => Set(Trans q)  -> Set q -> Sym -> Set q
transition ts qs s  = fromList [ q' | (q, t, q') <- toList ts, t == s, q `member` qs ]
```

We now want to consider how the machine reacts when we type a string, a *sequence of symbols*. An alternative picture, taking the human typist out of the loop, is to view an FSM as a machine with an input tape with a list of symbols. At each step the machine reads a symbol from the tape and makes a transition accordingly (if you are still thinking of the keyboard, it makes a transition as if the corresponding key had been pressed). At the next step it reads the next symbol; then moves – and so on .... When there are no symbols left, the machine halts.

Starting from a given set of states and consuming a string, one character at a time, takes us to a final set of states. We can write a recursive function to compute this final set of states

```
-- applying transitions for a string of symbols
final ::  Ord q => Set(Trans q) ->Set q -> [Sym] -> Set q
final ts ss [] = ss
final ts ss (a : as) = final ts (transition ts ss a) as
```

or we may use `foldl`

```
final ts = foldl (transition ts)
```

We say an FSM **accepts** a string `cs` if, when we start from its set of starting states, and consume the string, the final set of states includes an accepting state – that is to say, if the intersection of the final set of states with the set of accepting states is not empty.

We don't really need the function `final` ; using `foldl` we can just write:

```
accepts :: (Ord q) => FSM q -> [Sym] -> Bool
accepts (FSM _ _ ts ss fs)  string = (not.null) (fs /\ final)
  where final = foldl (transition ts) ss string
```

Note that, `foldl` works for sets as well as lists, (techically, this is because sets are `Foldable`).
`transition ts :: Set q -> Sym -> Set q` gives the transition for a single symbol, so that,
`foldl (transition ts) :: Set q -> [Sym] -> Set q` gives the transition for a string of symbols.

7. *No submission.* Use Haskell to check your answers to question 1.

   We can trace the action of the machine reading a string of symbols from the tape, by recording, at each step, which lights are on, to produce a list of sets of states. We call this list the trace of the computation.

8. Complete the definition of a function to produce the trace, by replacing each occurrence of `undefined` in the code.

```
trace :: Ord q -> FSM q -> [Sym] -> [Set q]
trace (FSM qs as ts ss fs) word = reverse (tr ss word)  where
  tr ss [] = undefined
  tr ss (w : ws) = undefined
```

   This function should first record the starting set of active states (the starting states); then, each time the machine consumes a symbol from the tape record the resulting set of active states. When the tape is empty the final set of lights is recorded.

   Given a string of length n the trace will be a list of length $n + 1$.

   The last entry in the trace tells us which lights are on, once every symbol from the string has been consumed. The machine accepts the string if one or more of these lit states is an accepting state.

9. *Optional* The reverse of an FSM is obtained by reversing the direction of each transition, and exchanging the sets of starting and finishing states. Complete this function to produce the reverse of a given FSM.

   Replace the `undefined` to complete this definition:

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

   Hint: define a function to reverse a transition and use `mapS` .

   How are the strings `reverseFSM m` accepts related to those accepted by the original machine, `m` ?

* * * * * *

FSM provide a simple model of computation. Each FSM answers some question about strings - the answer to the question is `yes` if it accepts the string and `no` if it does not. We will, soon, characterise exactly which questions about strings can be answered by a FSM. A regular expression (regex) is a kind of pattern. We will see next week that for each FSM there is a regex such that the strings accepted by the machine are exactly the strings matching that regex. Furthermore, for each regex there is a corresponding FSM.

3.2 Converting an N-FSM to a D-FSM

The idea of this construction in terms of our picture of a machine that displays various sets of lights, is to take each active set of lights reachable from the starting set of state, as a state of a new machine. If we had a machine with three lights, then a state of these three lights would correspond to a region in the Venn diagram, and we could construct a new machine with (at most) eight lights to achieve the same computations as the original.

This conversion makes any FSM into a DFA. It is called the "powerset construction." The machine is constructed as follows:

- The states of the DFA will be "superstates" of the original—each superstate is a *set* of states of the original machine. We need to include each set of active states reachable from the starting set of states.
- The alphabet of symbols is unchanged.
- The DFA will have a transition from superstate `superq` to superstate `superq'` whenever each state in `superq'` is the target of some state in `superq`.
- The accepting (super)states of the DFA are those which contain some accepting state of the original FSM.
- The initial (super)state is just the singleton set containing only the initial state of the original FSM.

Figure 2 shows two FSMs, one where the states are identified by integers, `m1 :: FSM Int` , and one where the states are characters, `m2 :: FSM Char` .

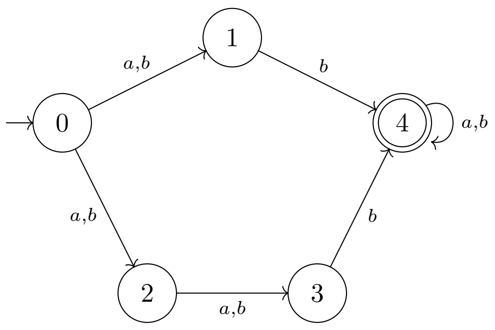
For instance, converting `m1 :: FSM Int` in Figure 2 yields `dm1 :: FSM (Set Int)` in Figure 3. (Haskell shows the set $\{1, 2, 3\}$, for example, as `fromList[1,2,3]` .)

Note how we take advantage of the fact that an FSM can have states of any type: the states of the FSM are integers and the states of the corresponding DFA are sets of integers; superstates are represented explicitly. (This is exactly why we made the type of an FSM parameterized by the type of the state.)

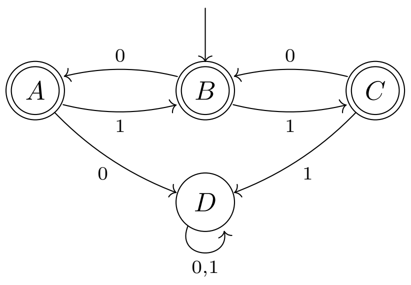
10. Complete the definition of this function so that given an FSM, a source superstate, and a symbol, it returns the target superstate.

    ```
    ddelta :: (Ord q) => FSM q -> (Set q) -> Char -> (Set q)
    ddelta (FSM  qs as ts ss fs ) source  sym = undefined
    ```

    The *target superstate* for a source superstate `source` , and a symbol, `sym` , is the set of states, `t` , such that, for some state, `s` in `source` , the machine has a `sym` -labelled transition from `s` to `t` . For example,

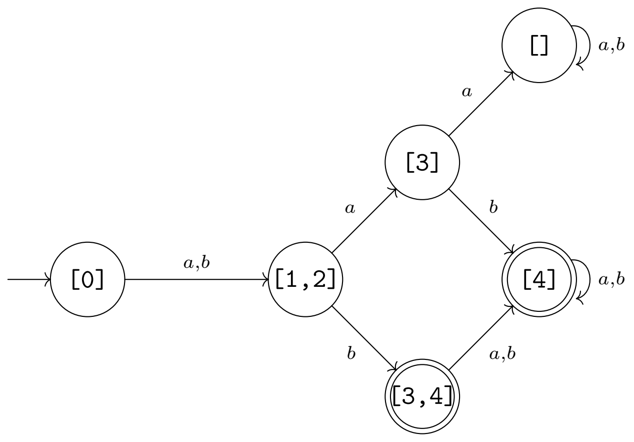


```
m1 :: FSM Int
m1 = mkFSM
     [0,1,2,3,4] -- states
     "ab"        -- symbols
     [ (0,'a',1), (0,'b',1), (0,'a',2), (0,'b',2), (1,'b',4)
     , (2,'a',3), (2,'b',3), (3,'b',4), (4,'a',4), (4,'b',4) ]
     [0]  -- starting
     [4]  -- accepting
```



```
m2 :: FSM Char
m2 = mkFSM
     "ABCD"      -- states
     "01"        -- symbols
     [('A', '0', 'D'), ('A', '1', 'B'), ('B', '0', 'A'), ('B', '1', 'C'),
      ('C', '0', 'B'), ('C', '1', 'D'), ('D', '0', 'D'), ('D', '1', 'D')]
     "B"   -- starting
     "ABC" -- accepting
```

Figure 2: Two finite state machines



```
dm1 :: FSM [Int]
dm1 = mkFSM
      [[],[0],[1,2],[3],[3,4],[4]] -- states
      "ab"                         -- symbols
      [([],   'a',[]),    ([],   'b',[])
      ,([0],  'a',[1,2]), ([0],  'b',[1,2])
      ,([1,2],'a',[3]),   ([1,2],'b',[3,4])
      ,([3],  'a',[]),    ([3],  'b',[4])
      ,([3,4],'a',[4]),   ([3,4],'b',[4])
      ,([4],  'a',[4]),   ([4],  'b',[4])]
      [[0]]        -- starting
      [[3,4],[4]] -- accepting
```

Figure 3: DFA corresponding to an FSM

```
*Tutorial9> ddelta m1 (fromList[0]) 'b'
fromList [1,2]
*Tutorial9> ddelta m1 (fromList[1,2]) 'b'
fromList [3,4]
*Tutorial9> ddelta m1 (fromList[3,4]) 'b'
fromList [4]
```

Hint: The transition is computed by applying the `transition` function defined above.

11. If the FSM has n states, then there are 2^n possible superstates that might appear in the DFA, but we need not consider all of these. We only care about the superstates that are reachable from the start state. In the next two questions, we'll compute which states are reachable. Complete the function `next` so that, given an FSM and a list of superstates, it returns the set of superstates that can be reached in a single transition from any of these.

```
next :: (Ord q) => FSM q -> Set(Set(q)) -> Set(Set(q))
next (FSM qs as ts ss fs) supers =
  fromList [ undefined | super <- toList supers, sym <- toList as ]
```

Hint: The value can be computed by applying `ddelta` to each superstate in the set and each symbol in the alphabet.

For example, the value

```
*Tutorial9> next m1 (fromList[fromList[0],fromList[1,2]])
fromList [fromList [1,2],fromList [3],fromList [3,4]]
```

can be computed from the following calls

```
*Tutorial9> ddelta m1 (fromList[0]) 'a'
fromList[1,2]
*Tutorial9> ddelta m1 (fromList[0]) 'b'
fromList[1,2]
*Tutorial9> ddelta m1 (fromList[1,2]) 'a'
fromList[3]
*Tutorial9> ddelta m1 (fromList[1,2]) 'b'
fromList[3,4]
```

Further hint: Use a comprehension with two generators to apply `ddelta` to each superstate in the input list of superstates, and to each symbol in the alphabet. (Look again at the definition of `cartesianProduct` given in Section 3.)

12. *Optional* Complete the definition of the function so that, given an FSM and a list of superstates, it returns the list of every superstate that can be reached by applying any (natural) number of transitions to some superstate in the list. (N.B. Zero is a natural number.) (Observe that we again use an @-pattern to bind `fsm` , so you can call any function that takes the machine as a parameter.)

```
reachable :: (Ord q) => FSM q ->  Set(Set(q)) -> Set(Set(q))
reachable fsm@(FSM qs as ts ss fs) supers =
  let new = undefined
  in if null new then supers else reachable fsm (supers \/ new)
```

For example

```
*Tutorial9> reachable m1 (fromList[fromList[0]])
fromList [fromList [],fromList [0],fromList [1,2],fromList [3],fromList [3,4],fromList [4]]
```

Hint: The value of the call above is computed starting with the following sequence of calls to `next`.

```
*Tutorial9> next m1 (fromList[fromList[0]])
fromList [fromList [1,2]]
*Tutorial9> next m1 (fromList[fromList[0], fromList[1,2]])
```

```
fromList [fromList [1,2],fromList [3],fromList [3,4]]
*Tutorial9> next m1 (fromList[fromList[0], fromList[1,2], fromList[3], fromList[3,4]])
fromList [fromList [],fromList [1,2],fromList [3],fromList [3,4],fromList [4]]
```

In general, one repeatedly applies `next` to extend the list until there is no further change. At each stage you can check whether `next` has produced any new superstates. You may use the set difference operation, \\ and `null` to do this. If there are no new superstates you are done – otherwise add these to your list and continue.

Notice that if we start from the list containing just the initial superstate, `reachable` will return every superstate that is reachable in the equivalent DFA.

13. *Optional* Complete this function so that it takes an FSM and a set `supers` of superstates and returns the subset (of `supers`) of accepting states.

```
dfinal :: (Ord q) => FSM q -> Set(Set(q)) -> Set(Set(q))
dfinal  fsm@(FSM qs as ts ss fs) supers = undefined
```

Hint: The logical functions `or and any all` that we've used with lists of things also work with sets of things; you can also use `filterS` . The functions `filterS` ,`mapS` etc. that work for sets, pretty much the `Prelude` versions of `filter` , and `map` do for lists

Remember that a superstate is final if it includes a final state of the original FSM. Write a function that given a superstate determines whether it contains a final state. Then use `S.filter` to select all final superstates from the set.

Example,

```
*Tutorial9> dfinal m1 (fromList[fromList[],fromList[0],fromList[1,2],fromList[3],fromList[3,4],fromList[4]])
fromList [fromList [3,4],fromList [4]]
```

14. *Optional* Complete this function which should take an FSM and a list of superstates and, for each symbol in the alphabet of the FSM, return a transition for each superstate in the list.

```
dtrans :: (Ord q) => FSM q -> Set(Set q) -> Set(Trans (Set q))
dtrans fsm@(FSM qs as ts ss fs) supers =
  fromList [ (q, s, undefined) | q <- toList supers, s <- toList as ]
```

For example,

```
*Tutorial9> dtrans m1(fromList[fromList[],fromList[0],fromList[1,2],fromList[3],fromList[3,4],fromList[4]])
fromList [(fromList [],'a',fromList []),(fromList [],'b',fromList[]),(fromList [0],'a',fromList [1,2])
,(fromList [0],'b',fromList [1,2]),(fromList [1,2],'a',fromList[3]),(fromList [1,2],'b',fromList [3,4])
,(fromList [3],'a',fromList []),(fromList [3],'b',fromList[4]),(fromList [3,4],'a',fromList [4])
,(fromList [3,4],'b',fromList [4]),(fromList [4],'a',fromList [4]),(fromList [4],'b',fromList [4])]
```

Hint: The target of each transition can be computed using `ddelta` . For example,

```
*Tutorial9> ddelta m1 (fromList[0]) 'a'
fromList [1,2]
*Tutorial9> ddelta m1 (fromList[0]) 'b'
fromList [1,2]
*Tutorial9> ddelta m1 (fromList[1,2]) 'a'
fromList [3]
*Tutorial9> ddelta m1 (fromList[1,2]) 'b'
fromList [3,4]
```

Further hint. Use a comprehension with two generators, one iterating over the list of superstates and one iterating over the alphabet.

15. *Optional* Complete this function to take an FSM and return the corresponding DFA.

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

For example, `deterministic m1` returns `dm1` .

Hint: Use `reachable` to compute the set of states, use the same alphabet as the given FSM, use as the start state the superstate containing only the start state of the FSM, use `dfinal` to compute the final states, and `dtrans` to compute the transition.