# CL exercise for Tutorial 9

## Introduction

### Objectives

This tutorial lets you play with finite state machines, using the FSM workbench, and start on implementing finite state machines in Haskell.

### Tasks

Exercise 1 has no submission.
Exercise 2, 3, and 4 are mandatory, with code answers.
Exercise 5 is optional, with written answers.

### Submission

Submit the file `CLTutorial9.hs` with your Haskell code for exercises 2–4.

For exercise 5, you may either type your answer in the comment section at the very end of the `CLTutorial9.hs` file, or you may submit a file `cl-tutorial-9` (image or pdf).

### Deadline

16:00 Tuesday 22 November

### Reminder

#### Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit.

You can find guidance at the School page

https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

## The FSM workbench

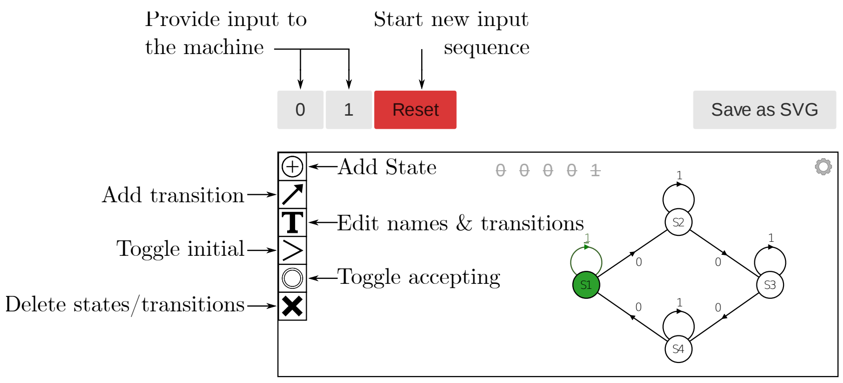
The FSM workbench, designed and implemented by Matthew Hepburn, is an interactive tool for designing and simulating machines. You may find it useful for testing your ideas. The workbench provides tools for creating, editing, and simulating finite state machines.

The 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.

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.



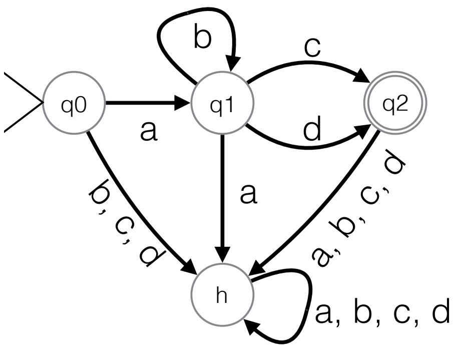
## Exercise 1 –not marked–

Work through the first half of the exercises on the workbench – click on **Exercises** to start – ending with the Vending Machine example.

*No submission for this exercise.*

## Exercise 2 –mandatory—marked–

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



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

1. `"abbd"`
2. `"ad"`
3. `"aab"`
4. `"abbbc"`
5. `"ac"`

Give your answer by deleting the strings that are *not* accepted from the list `ex2strings` in your Haskell file.

(1), (2), (4), (5)

## FSMs in Haskell

We will now start working on the Haskell implementation of FSMs. Because we want to use the same code next week, the datatype will be a bit more general than the formalism in week 9 lectures:

- we will have a *set* $S$ of start states instead of a single start state $q_0$
- transitions will be a *relation* $\delta \subseteq Q \times \Sigma \times Q$ instead of a function $Q \times \Sigma \to Q$. We more or less have to do this anyway, as we can only represent a function by a list or set of its arguments–value tuples.[1] So instead of writing $\delta(q, a) = q'$ we write $(q, a, q') \in \delta$ (or sometimes $\delta(q, a, q')$).

We'll see next week how to understand these more general automata, but for this week we will work only with DFAs.

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 is represented by the set $\delta$ of labelled transitions.

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.

[1]Yes, we could build a datatype that took actual Haskell functions as transition functions, but that would be harder to compute with.

```haskell
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 is 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` :

```haskell
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:

```haskell
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, $Q$;

`as :: Set Sym` the *alphabet* of symbols, $\Sigma$;

`ts :: Set (q,Sym,q)` the *transitions*, $\delta$;

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

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

The automaton from exercise 2 can be coded thus, where state $h$ is 3:

```haskell
eg1 = mkFSM [0..3] "abcd"
   ([(0,'a',1)]++(map (\x->(0,x,3)) "bcd")
     ++[(1,'b',1),(1,'c',2),(1,'d',2),(1,'a',3)]
     ++(map (\x->(2,x,3)) "abcd")
     ++(map (\x->(3,x,3)) "abcd"))
   [0] [2]
```

Notice the use of map and lambda to reduce boring typing for the black hole transitions.

The ultimate goal is to write the Haskell function that implements the powerset construction that converts any FSM to a DFA. However, we won't look at this until next week, so this tutorial lays some of the groundwork.

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 the following questions, we give you a template for the answer, with some `undefined` terms. Your task is to replace each occurrence of `undefined` with appropriate code.

## Exercise 3 –mandatory—marked–

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

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) = (size ss == undefined)
  && undefined [ length[ q' | q' <- qs, (q, a, q')`member`ts ] == 1
                | q <- toList qs, a <- toList as ]
```

```
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 ]
```

This more general FSM is a DFA exactly if it has one start state and for each $q$ and $a$, there is exactly one $q'$ such that $\delta(q, a, q')$.

## Running a machine

Now we introduce Haskell functions that define how an FSM runs. In lectures so far, we have marked the current, or active, state in green. In the more general FSMs we'll look at next week, there can be several 'active states'. If the currently active states are $A$, and the input is $a \in \Sigma$, then the next set of active states is just

$$\{q' : \exists q \in A.(q, a, q') \in \delta\}$$

The function transition translates this mathematical definition into Haskell to define how the active states 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 ]
```

Now let's consider running through a string of inputs, as on slide 13.

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.

We can trace the action of the machine reading a string of input symbols by recording, at each step, which states are active, to produce a list of sets of states. We call this list the trace of the computation. (On slide 13, working only with DFAs, the trace was a list of states; now it is a list of sets of states.)

## Exercise 4 –mandatory—marked–

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 = tr ss word where
  tr ss' [] = undefined
  tr ss' (w : ws) = undefined
```

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

This function should first record the starting set of active states (the starting states); then, each time the machine consumes a symbol from the input, record the resulting set of active states. When the input is empty the final set of active states 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 states are active, once every symbol from the string has been consumed. The machine accepts the string if one or more of these active states is an accepting state.

## Exercise 5 –optional—marked–

Fix an input alphabet $\Sigma$. Show that the set $\mathscr{L}$ of regular languages over $\Sigma$, together with the operations of complement, intersection, and union, is a boolean algebra.

We know from lectures that the set is closed under complement, intersection and union, so those operations give the boolean operations, with $\varnothing$ and $\Sigma^*$ as $\mathbf{0}$ and $\mathbf{1}$.

The set $\mathscr{M}$ of DFAs over $\Sigma$ with the operations $^-, \times, +_\mathrm{d}$ is not a boolean algebra. Why not?

Because we don't have a useful notion of equality on DFAs, and so there's no single element $\mathbf{0}$, for example, and no reason that $M + (M' \times M'')$ should $= (M \times M') + (M \times M'')$.

If you know about equivalence relations and equivalence classes (or are willing to look them up), what is an equivalence relation $\approx$ on $\mathscr{M}$ that makes $\mathscr{M}/\approx$ a boolean algebra?

$M \approx M'$ iff $L(M) = L(M')$