Finite State Machines
Informatics 1 – Introduction to Computation
Functional Programming Tutorial 8

Week 9 (12–16 Nov.)

Each tutorial involves a mixture of individual homework as a preparation step, followed by work and discussion done by groups of five to six students during the tutorial meeting. You are being encouraged to work together and help each other throughout the whole tutorial. A number of tutors will also be available to give help or advice. If at any point your group needs assistance please raise your hand to draw our attention.

Please go through the entire worksheet in advance of the tutorial; answer the questions when needed or take some personal notes. Make sure that you complete the *Answer Sheet* of the third section, which in turn corresponds to various tasks throughout the tutorial. Don't forget to bring your answers along with any personal notes — personal notes may include printouts of code and test results.

Assessment is formative, meaning that marks from coursework do not contribute to the final mark. But coursework is not optional. If you do not do the coursework you are unlikely to pass the exams.

Attendance at tutorials is obligatory; please send email to Rob.Armitage@ed.ac.uk if you cannot join your assigned tutorial.

1 Homework: Finite State Machines in Haskell

In the Computation & Logic part of the course, you've learned about finite state machines (FSMs), both deterministic (D-FSMs) and nondeterministic (N-FSMs), and you've learned how the latter can be transformed into the former.

We will begin with FSMs without ε-transitions and with a single start state, and later consider how to generalise.

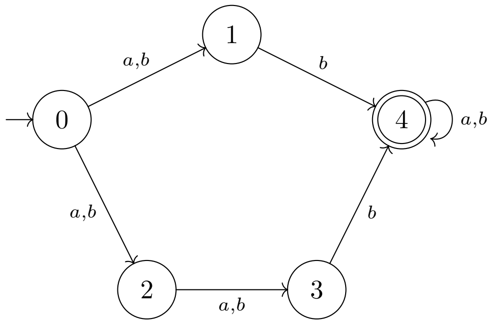
1.1 Finite State Machines over arbitrary states

Here is the type we'll use for FSMs whose states have type `q`, where `q` might be any type:

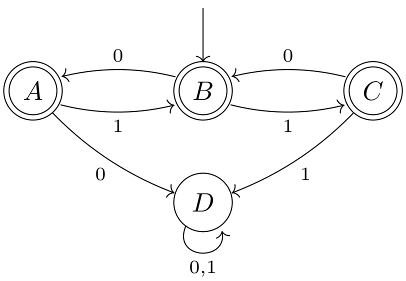
```
type FSM q = ([q], Alphabet, q, [q], [Transition q])
type Alphabet = [Char]
type Transition q = (q, Char, q)
```

In this assignment, a FSM is a five-tuple `(k,a,s,f,t)`, consisting of: the universe of all states (`k`, a list of states), the alphabet (`a`, a list of characters), the start state (`s`, a state), the final states (`f`, a list of states), and the transitions (`t`, a list of transitions). Each transition `(q,x,q')` has a source state `q`, a symbol `x`, and a target state `q'`. (This is related to, but a little different from, the `Haskell` representation of DFMs you used in CL tutorial 2.)

Figure 1 shows two FSMs, one where the states are identified by integers, `m1 :: FSM Int`, and one where the states are characters, `m2 :: FSM Char`.




```
m1 :: FSM Int
m1 = ([0,1,2,3,4],
      ['a','b'],
      0,
      [4],
      [(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)])
```




```
m2 :: FSM Char
m2 = (['A','B','C','D'],
      ['0','1'],
      'B',
      ['A','B','C'],
      [('A', '0', 'D'), ('A', '1', 'B'),
       ('B', '0', 'A'), ('B', '1', 'C'),
       ('C', '0', 'B'), ('C', '1', 'D'),
       ('D', '0', 'D'), ('D', '1', 'D')])
```

Figure 1: Two finite state machines

Exercise 1

(a) Define five functions to retrieve the five components of a machine.

```
states :: FSM q -> [q]
alph   :: FSM q -> Alphabet
start  :: FSM q -> q
final  :: FSM q -> [q]
trans  :: FSM q -> [Transition q]
```

For example,

```
*Main> states m1
[0,1,2,3,4]
*Main> final m2
"ABC"
```

Hint: Use a pattern `(k,a,s,f,t)` as the argument of each function.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions `[1a]`.

Exercise 2

(a) Write a function that given an FSM, a source state, and a symbol, returns a list of all states that are the target of a transition for the given source state and symbol.

```
delta :: (Eq q) => FSM q -> q -> Char -> [q]
```

(The type declaration has a clause `(Eq q)` because you will need to use equality `(==)` to compare states.) For example,

```
*Main> delta m1 0 'a'
[1,2]
*Main> delta m2 'B' '0'
"A"
```

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[2a]`.

Exercise 3

(a) Write a function that given an FSM and a string returns `True` when the FSM accepts the string. The function should work with any FSM, deterministic or otherwise.

```
accepts :: (Eq q) => FSM q -> String -> Bool
```

For example,

```
*Main> accepts m1 "aaba"
True
*Main> accepts m2 "001"
False
```

Hint: Here is a skeleton of the function definition:

```
accepts :: (Eq q) => FSM q -> String -> Bool
accepts m xs   =  acceptsFrom m (start m) xs

acceptsFrom :: (Eq q) => FSM q -> q -> String -> Bool
acceptsFrom m q []      =  q `elem` final m
acceptsFrom m q (x:xs)  =  ...
```

The function `acceptsFrom` returns true if and only if it accepts the given string starting in the given state. For example, machine `m1` in its start state accepts the string `"aab"`.

```
*Main> acceptsFrom m1 0 "aab"
True
```

We previously saw that

```
*Main> delta m1 0 'a'
[1,2]
```

Hence, from state `0`, on seeing the symbol `'a'`, the machine can move to either of state `1` or state `2`. We can recursively use `acceptsFrom` to determine if the remaining string `"ab"` is accepted in either of these states.

```
*Main> acceptsFrom m1 1 "ab"
False
*Main> acceptsFrom m1 2 "ab"
True
```

Since the remaining string is accepted in at least one of the subsequent states, the original call succeeds.

Further hint: To fill in the `...`, use a list comprehension that iterates over the states returned by `delta` and uses `acceptsFrom` recursively to compute a list of booleans; then use an appropriate function to combine the booleans.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[3a]`.

1.2 Converting an N-FSM to a D-FSM

To convert an N-FSM into a D-FSM, we can use a technique called the "powerset construction." The machine is constructed as follows:

- The states of the D-FSM will be "superstates" of the original—each superstate is a *set* of states of the original machine.
- The D-FSM 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 D-FSM are those which contain some accepting state of the original N-FSM.
- The initial (super)state is just the singleton set containing only the initial state of the original N-FSM.

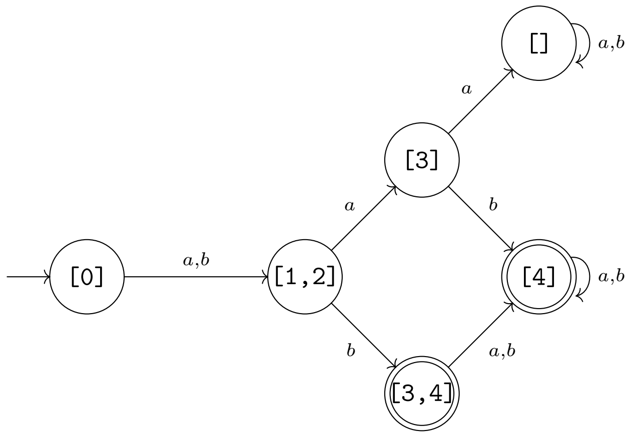
For instance, converting the first N-FSM in Figure 1 yields the D-FSM in Figure 2.

```
m1  :: FSM Int
dm1 :: FSM [Int]
```

Note how we take advantage of the fact that an FSM can have states of any type: the states of the N-FSM are integers and the states of the corresponding D-FSM are lists of integers, so superstates can be represented explicitly. (This is exactly why we made the type of an FSM parameterized by the type of the state.) Our goal is to write a Haskell function that given `m1` as input will produce `dm1` as output.

Exercise 4

(a) We use lists to represent sets. So that it is easy to compare sets, we will always represent sets by a canonical list that contains the states *in order* with *no duplicates*. Write a function that converts a list of states to its canonical form.




```
dm1 :: FSM [Int]
dm1 =  ([[],[0],[1,2],[3],[3,4],[4]],
        ['a','b'],
        [0],
        [[3,4],[4]],
        [([],   '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])])
```

Figure 2: D-FSM corresponding to an N-FSM

```
canonical :: (Ord q) => [q] -> [q]
```

(The `(Ord q)` clause is in the type because you will need to assume there is some order on the states.) For example,

```
*Main> canonical [1,2]
[1,2]
*Main> canonical [2,1]
[1,2]
*Main> canonical [1,2,1]
[1,2]
```

Hint. Use the library functions `List.sort` and `List.nub`.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[4a]`.

Exercise 5

(a) Write a function that given an N-FSM, a source superstate, and a symbol, returns the target superstate.

```
ddelta :: (Ord q) => FSM q -> [q] -> Char -> [q]
```

The *target superstate* is the set of states to which the machine can move, starting from one of the source states, given the input symbol. For example,

```
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'b'
[3,4]
*Main> ddelta m1 [3,4] 'b'
[4]
```

Important: The target superstate should be given in its canonical form.

Hint: The transition is computed by applying the delta function to each state in the given superstate and then combining and canonicalizing the results. For example, `ddelta m1 [0] 'b'` is computed from

```
*Main> delta m1 0 'b'
[1,2]
```

and `ddelta m1 [1,2] 'b'` is computed from

```
*Main> delta m1 1 'b'
[4]
*Main> delta m1 2 'b'
[3]
```

Further hint: Use a list comprehension and possibly the library function `concat`.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[5a]`.

Exercise 6

If the N-FSM has n states, then there are 2^n possible superstates that might appear in the D-FSM, 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.

(a) Write a function `next` that, given an N-FSM and a list of superstates, finds all of the superstates that can be reached in a single transition from any of these and adds these reachable superstates to the input list.

```
next :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

Each superstate must be canonical, and there should be no duplicates in the list.

For example,

```
*Main> next m1 [[0]]
[[0],[1,2]]
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
*Main> next m1 [[0],[1,2],[3],[3,4]]
[[],[0],[1,2],[3],[3,4],[4]]
*Main> next m1 [[],[0],[1,2],[3],[3,4],[4]]
[[],[0],[1,2],[3],[3,4],[4]]
```

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

```
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
```

can be computed from the following calls

```
*Main> ddelta m1 [0] 'a'
[1,2]
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'a'
[3]
*Main> ddelta m1 [1,2] 'b'
[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; don't forget to add the input list of superstates to the result, and make sure that no superstate is added twice.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[6a]`.

Exercise 7

(a) Write a function that given an N-FSM and a list of superstates adds to the list any other superstates that can be reached by applying any number of transitions to any superstate in the list.

```
reachable :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

For example

```
*Main> reachable m1 [[0]]
[[0],[1,2],[3],[3,4],[],[4]]
```

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

```
*Main> next m1 [[0]]
[[0],[1,2]]
*Main> next m1 [[0],[1,2]]
[[0],[1,2],[3],[3,4]]
*Main> next m1 [[0],[1,2],[3],[3,4]]
[[],[0],[1,2],[3],[3,4],[4]]
*Main> next m1 [[],[0],[1,2],[3],[3,4],[4]]
[[],[0],[1,2],[3],[3,4],[4]]
```

In general, one repeatedly applies `next` to extend the list until there is no further change. Notice that if we start from the list containing just the initial superstate, `reachable` will return every superstate that is reachable in the equivalent D-FSM.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[7a]`.

Exercise 8

(a) Write a function that takes a N-FSM and a list of superstates and returns a list of those that are final (accepting) in the D-FSM.

```
dfinal :: (Ord q) => FSM q -> [[q]] -> [[q]]
```

Remember that a superstate is final if it contains a final state of the original N-FSM. For example,

```
*Main> dfinal m1 [[],[0],[1,2],[3],[3,4],[4]]
[[3,4],[4]]
```

Hint: First write a function that given a superstate determines whether it contains a final state, using the `or` function and a comprehension. Then use it to select all final superstates from the list.

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[8a]`.

Exercise 9

(a) Write a function that takes a N-FSM and a list of superstates and returns a transition for each superstate in the list and each symbol in the alphabet of the N-FSM.

```
dtrans :: (Ord q) => FSM q -> [[q]] -> [Transition [q]]
```

For example,

```
*Main> dtrans m1 [[],[0],[1,2],[3],[3,4],[4]]
[([],'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])]
```

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

```
*Main> ddelta m1 [0] 'a'
[1,2]
*Main> ddelta m1 [0] 'b'
[1,2]
*Main> ddelta m1 [1,2] 'a'
[3]
*Main> ddelta m1 [1,2] 'b'
[3,4]
```

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

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned function `[9a]`.

Exercise 10

(a) Write a function that takes an N-FSM and returns the corresponding D-FSM.

```
deterministic :: (Ord q) => FSM q -> FSM [q]
```

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

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

(b) Fill out the *Answer Sheet* in Section 2 of the tutorial with the body of the aforementioned functions `[10a]`.

2 Answer Sheet

Please fill in your name as another member of your group will check your answers.

Name:

Exercise	

1a:
`states`
`alph`
`start`
`final`
`trans`	

2a: `delta`	
3a: `accepts`	
4a: `canonical`	
5a: `ddelta`	

6a: `next`	
7a: `reachable`	
8a: `dfinal`	
9a: `dtrans`	
10a: `deterministic`	

3 Tutorial Activities

Exercise 11

Compare each of your answers in the *Answer Sheet* with a buddy sitting next to you. Try to convince each other to agree on an answer and if it's possible use a computer to check your implementations.

Exercise 12

You will now modify the algorithm for converting an N-FSM to a D-FSM to take account of ε-transitions, where transitions between states can take place without consumption of input symbols. See Figure 3 for an example of an N-FSM with ε-transitions, and Figure 4 for the corresponding D-FSM.

Read the following material before the tutorial and try them out on that example so that you understand the functions described. During the tutorial, work with the person sitting next to you to define the functions in Haskell.

(a) Write a function that, given an N-FSM M with ε-transitions and a superstate qs of M, computes the "one-step ε-closure" of qs: the list of all states, including the states in qs themselves, that can be reached from states in qs via a single ε-transitions of M.

```
oneStepClose :: (Ord q) => FSM q -> [q] -> [q]
```

For example,

```
*Main> oneStepClose m3 [0]
[0,1,5]
*Main> oneStepClose m3 [1]
[1]
```

(b) Using `oneStepClose`, write a function that, given an N-FSM M with ε-transitions and a superstate qs of M, computes the "ε-closure" of qs: the list of all states, including the states in qs themselves, that can be reached from states in qs via only ε-transitions of M.

```
eClose :: (Ord q) => FSM q -> [q] -> [q]
```

Care is required to avoid non-termination when M contains ε-loops, as would happen if the transition `(5,epsilon,4)` were added to `m3` in Figure 3.

Hint: Here is a skeleton of the function definition:

```
eClose :: (Ord q) => FSM q -> q -> [q]
eClose m qs | qs == oneStepClose m qs = qs
            | otherwise               = ...
```

(c) Extend your definition of `ddelta` to take account of ε-transitions from the states in the target superstate.

```
eddelta :: (Ord q) => FSM q -> [q] -> Char -> [q]
```

Then, adapt your definitions of `next` and `dtrans` to use `eddelta` instead of `ddelta`, and adapt your definition of `reachable` to use `enext` instead of `next`.

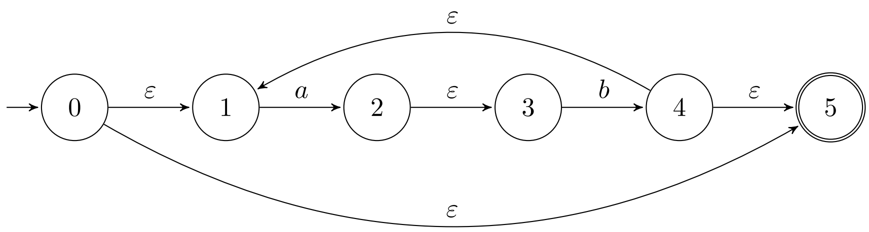
```
enext :: (Ord q) => FSM q -> [[q]] -> [[q]]
ereachable :: (Ord q) => FSM q -> [[q]] -> [[q]]
edtrans :: (Ord q) => FSM q -> [[q]] -> [Transition [q]]
```

(d) Finally, use these components to give a version of the function `deterministic` that converts an N-FSM with ε-transitions to a D-FSM.

```
edeterministic :: (Ord q) => FSM q -> FSM [q]
```

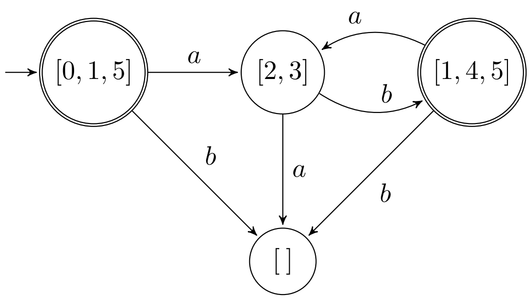
The initial superstate of the D-FSM should be the ε-closure of the superstate containing only the start state of the N-FSM, and its set of superstates is those that are reachable from there. The final superstates are the same as the ones in the construction without ε-transitions.

Check that `edeterministic m3` is `dm3`.




```
m3 :: FSM [Int]
m3 = ([0,1,2,3,4,5],
      ['a','b'],
      0,
      [5],
      [(0, epsilon, 1), (0, epsilon, 5),
       (1, 'a',     2),
       (2, epsilon, 3),
       (3, 'b',     4),
       (4, epsilon, 5), (4, epsilon, 1)])
```

Figure 3: An N-FSM with ε-transitions




```
dm3 :: FSM [Int]
dm3 =  ([[],[0,1,5],[1,4,5],[2,3]],
        ['a','b'],
        [0,1,5],
        [[0,1,5],[1,4,5]],
        [([],      'a', []),    ([],      'b', []),
         ([0,1,5], 'a', [2,3]), ([0,1,5], 'b', []),
         ([1,4,5], 'a', [2,3]), ([1,4,5], 'b', []),
         ([2,3],   'a', []),    ([2,3],   'b', [1,4,5])])
```

Figure 4: D-FSM corresponding to an N-FSM with ε-transitions

4 Optional Material

Please note that you may **NOT** have time to complete this last part during the tutorial; you can do it at home afterwards if you want.

In this section you are asked to implement a FSM which checks whether a string matches a regular expression. There are some QuickCheck properties already defined to help you along. For all of these questions, we use `['a'..'z']` as the alphabet.

Exercise 13

(a) Write a function `charFSM :: Char -> FSM Bool` that given a character returns an FSM that accepts a single instance of that character.

(b) Write a function `emptyFSM :: FSM ()` that returns an FSM that accepts no strings.

Exercise 14

The concatenation of two FSMs `A` and `B` with the same alphabet is an automaton which accepts an input word if `A` accepts some prefix of the input word and `B` accepts the rest of the word. Implement a function

```
concatFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (Either q q')
```

which returns the concatenation of the input FSMs. Your function should work on both D-FSMs and N-FSMs.

Exercise 15

(a) Write a function `intFSM :: FSM a -> FSM Int` that translates a `FSM q` (D-FSM or N-FSM) into an equivalent `FSM Int` which has a state space `[0..n]`.

(b) Write a function `stringFSM :: String -> FSM Int` that returns a FSM that accepts exactly the input string.

Exercise 16

Say that a FSM is *complete* if there is a transition from every state for each character of the alphabet.

(a) Write a function `completeFSM :: FSM a -> FSM (Maybe a)` that takes a FSM and returns an equivalent complete FSM.

Hint: Use `Nothing` to add the missing transitions.

(b) The union of two FSMs `A` and `B` is a FSM which accepts a word if either `A` or `B` accepts it. Write a function

```
unionFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (Maybe q, Maybe q')
```

which returns the union of the input FSMs. Your function should work on both D-FSMs and N-FSMs. Use `completeFSM` and `intFSM`.

Hint: A state of the union FSM consists of a pair `(q,q')` where `q` is a state of `A` and `q'` is a state of `B`. A transition in the union FSM is a triple `((q_0,q_1),char,(q_0',q_1'))` such that `(q_0,char,q_0')` and `(q_1,char,q_1')` are transitions in `A` and `B` respectively.

Exercise 17

(a) The Kleene star closure of a FSM `A` accepts the empty word and any input word which consists of a concatenation of words accepted by `A`.

Write a function `star :: FSM q -> FSM q` which returns the Kleene star closure of the input automaton. Your function should work on both D-FSMs and N-FSMs.

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

Exercise 18

In the last part of this tutorial, you will implement two additional operations, complement and intersection.

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

(b) The intersection of two FSMs `A` and `B` is a FSM which accepts a word if and only if both `A` and `B` accept it.
Implement a function

```
intersectFSM :: (Ord q, Ord q') => FSM q -> FSM q' -> FSM (q,q')
```

which returns the intersection of the input FSMs. Your function should work on both D-FSMs and N-FSMs.

(c) Use the QuickCheck properties at the bottom of tutorial8.hs to write your own tests.