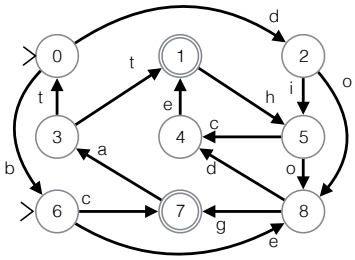
FSM (i)

1 FSM

This tutorial lets you play with finite state machines (FSM), also known as finite state automata (FSA).

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
- $S \subseteq Q$ start states
- $F \subseteq Q$ final (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.

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 $0 \xrightarrow{d} 2 \xrightarrow{o} 8 \xrightarrow{g} 7$.

1. (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?[2]

   (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 of length ≥ 3 as you can manage, but no non-words.

We introduce a Haskell type `FSM q`[3] to match our mathematical definition; `q` is the type of the , We use strictly-increasing, ordered lists to represent sets, as introduced in FP lectures 12-13. The function `set = nub.sort` is used to establish the invariant.

```
type Sym = Char
type Trans q = (q, Sym, q)
--   FSM states symbols transitions start final
data FSM q = FSM [q] [Sym] [Trans q] [q] [q] deriving Show
mkFSM qs as ts ss fs = -- use this when creating an FSM, to establish set invariants
  FSM (set qs) (set as) (set ts) (set ss) (set fs)
```

Our example may be encoded as follows:

```
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"
```

[1] Our formulation differs from that found in many elementary texts:

- Some texts describe a transition function $:: Q \times \Sigma \longrightarrow \wp(Q)$.
  This is just the function that acts as $(q, \sigma) \longmapsto \{q' \mid (q, \sigma, q') \in \Delta\}$.
- 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.

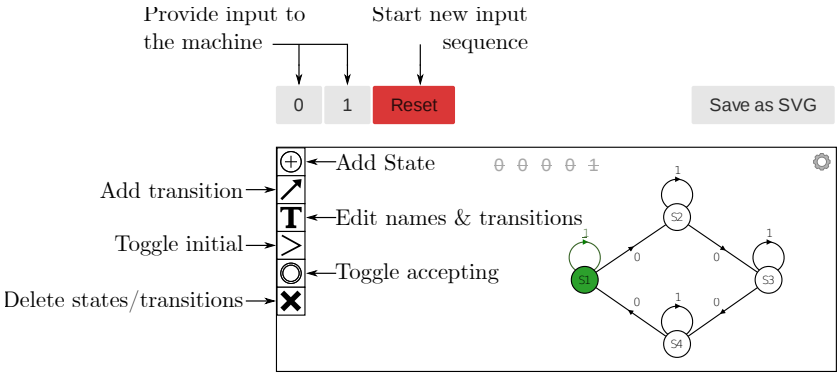
[2] Try this, but don't worry if you don't succeed – you will soon have the tools to do this.

[3] This is not the representation used in FP9 tutorial; we are using a slightly different formulation,but it is easy to translate each to the other.

2 The FSM workbench

The FSM workbench, designed and iplemented 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.

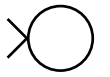
You can toggle each tool on/off by clicking it. When no tools are active you can drag the states of your FSM to rearrange the layout.



Your first task is to work through some exercises on the workbench – click on **Exercises** to start.

1. Work through the first half of the exercises, ending with 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.

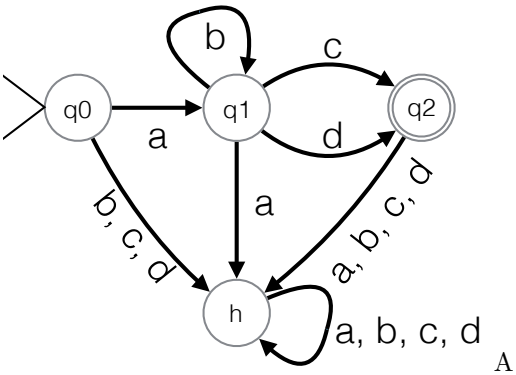


Initial

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.

2. 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).

2.1 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 $Q \times \Sigma \longrightarrow Q$ which means that for each state,symbol pair, $(q, \sigma) \in Q \times \Sigma$, there is at exactly one q' such that $(q, \sigma, q') \in \Delta$.

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

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.*

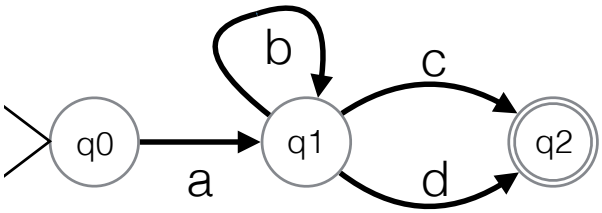
We call such a state a *black hole* state.

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.

2.2 Running an FSM in Haskell

In this section we introduce Haskell functions that define how an FSM runs.[4] 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 button for each symbol $\sigma \in \Sigma$. Pressing a button, σ, (we may only press one at a time) changes the lights. A state, q', is lit after the button-press iff there is an arrow $q \xrightarrow{\sigma} q'$, from some state q that was lit before, to q', labelled with σ . The next set of lit states is given by,

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

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

```
move :: (Ord q) => [Trans q] -> Sym -> [q] -> [q]
move ts x ss = [ q' | q' <- qs, or[(q,x,q')`elem`ts]| q <- ss] ]
```

Observe that `step fsm x` satisfies the set invariants, provided `fsm` does.

Following a button-press we get a new machine – only the start-states have changed:

[4] We will normally use a specific variable for each component of the machine:

`qs :: [q]` the states, Q;
`ss :: [q]` the *start*, states, S;
`as :: [Sym]` the *alphabet* of symbols, Σ;
`ts :: [(q,Sym,q)]` the *transitions*, Δ;
`fs :: [q]` the *final*, or accepting, states, Q.

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

We have introduced the machine inputs as buttons. However, we also want to consider how the machine reacts to a *sequence of inputs*. For this, we picture 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 moves accordingly (if you are still thinking of the button picture, it moves as if the corresponding button 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.

We trace the action of the machine reading a string of symbols from the tape, by recording, at each step, which lights are on.

```
trace :: Ord q => FSM q -> [Sym] -> [[q]]
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs
trace     (FSM _ _ _ ss _) []     = [ss]
```

At each step, this function records the set of lit lights, the machine consumes a symbol from the tape and moves on. When the tape is empty the final set of lights is recorded.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.

```
accepts fsm@(FSM qs as ts ss fs) string =
  or[ q`elem`fs | q <- last (trace fsm string) ]
```

We can write the function `accepts` directly:

```
accepts :: (Ord q) => FSM q -> String -> Bool
accepts fsm (x : xs) = accepts (step fsm x) xs
accepts (FSM _ _ _ ss fs) "" = or[ q`elem`fs | q <- ss ]
```

3. To get started, define some basic finite state machines, code them in Haskell and test them to check that they accept the language specified:

   (a) a machine that accepts only a string consisting of a single, given character.

   ```
   charFSM :: Char -> FSM Bool
   ```

   (b) a machine that accepts only a given string.

   ```
   stringFSM :: String -> FSM Int
   ```

   (c) a machine that accepts only the empty string `""`

   ```
   emptyFSM :: FSM Int
   ```

   (d) a machine that accepts nothing

   ```
   nullFSM :: FSM ()
   ```

4. This question concerns the *simple DFA* introduced above. We have two represntations of this machine – with and without the black hole convention. You will implement each in Haskell and compare their behaviour.

   (a) Represent each machine, as a Haskell value, `simpleFSM` and, with the black hole convention, `simpleFSMbh`. The first will have four states and 16 transitions, while the second has three states and four transitions.

   (b) For each machine inspect the traces for various input strings – some accepted some not.

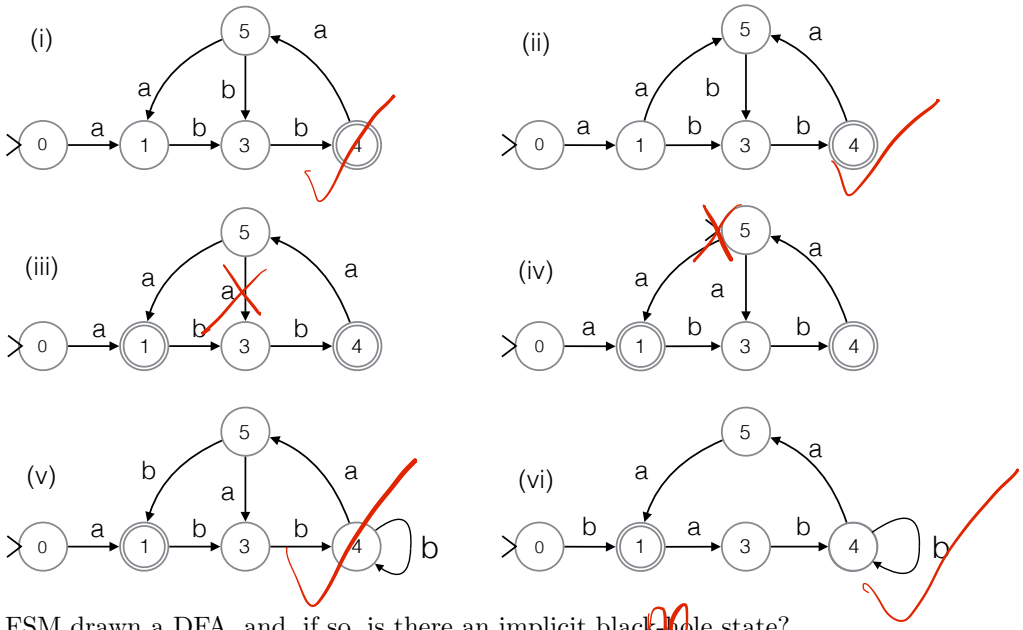
   How do the traces compare, in particular, when the DFA is in its black hole state, what happens to the reduced machine using the black hole convention?

5. (a) The code for the machine `eg1` is included in the `CL7.hs` file. Use Haskell to test the words you found in 1a.

   (b) Code your machine from 1c in Haslell, and check your words.

   Can you be sure that your machine accepts *only* these words?

6. For each of the six FSMs in the diagram, answer the questions given below.



(a) Is the FSM drawn a DFA, and, if so, is there an implicit black-hole state?

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

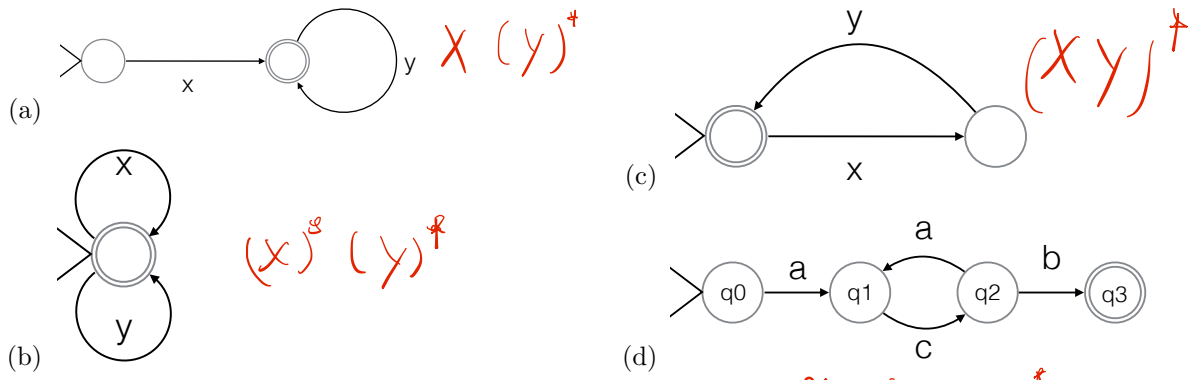
i. `abb`	iii. `ab`	v. `babaaab`	vii. `abbbbab`
ii. `abbaa`	iv. `aabbab`	vi. `aabb`	viii. `abbaabb`

(c) Code each machine as an FSM in Haskell and test your answers.

(d) Try[5] to give a regular expression that matches the strings accepted by the machine.

The following questions use the black hole convention - and you can use it in your answers.

7. Give a regex that describes the strings accepted by each of the following DFA.



[5] **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.

8. Draw an FSM that accepts the strings described by each of the following regex.[6]
In each case say whether your FSM is a DFA.

(a) `a(bb)*d`	(c) `ab*|d*`	(e) `a(b|c)d`	(g) `a*(b|cd)*`	(i) `a(b|d)*`
(b) `a(b*|d*)`	(d) `a*b|cd*`	(f) `(ab)*d`	(h) `(a|b)*d*`	(j) `a(b*d*)*`

Remember that an FSM may have any number of accepting states.

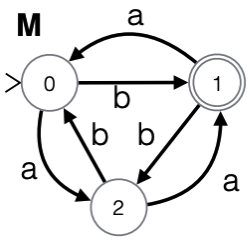
9. The reverse of an FSM, M, accepts the reverse of any string accepted by M. Write a function

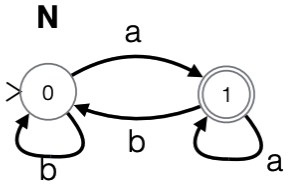
```
reverseFSM :: Ord q => FSM q -> FSM q
```

and use `quickCheck` to test it with the following property

```
prop_reverse :: String -> Bool
prop_reverse s = accepts (reverseFSM $ stringFSM s) (reverse s)
```

We consider the following two DFAs for question 10 and 11:





10. The product of two DFA accepts the intersection of the languages accepted by each machine.
    (a) Use **pen and paper**, produce a product machine of **M** and **N**.
    (b) Is the product of two DFA a DFA? Explain your answer.
11. For a DFA we can define a sum machine which is just like the product machine defined in Question 2.2, except that it has a different set of accepting states.
    (a) Use **pen and paper**, produce a sum machine of **M** and **N**. What have changed comparing to the product machine?
    (b) Find a pair of FSM (at least one of which will be non-deterministic) such that `sumDFA fsm1 fsm2` does not recognise the union of the languages recognised by the two FSM.

* * * * * *

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 kind of pattern, It turns out 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.

[6] In this section we use basic regular expressions, with alphanumeric symbols `a-z, 0-9`; parentheses `( )` always mean grouping. regex are contructed from symbols using three basic operations. In order of decreasing precedence, these are: iteration `*`, concatenation, and alternation `|` .
So `a*b|cd*` means `((a*)b)|(c(d*))`, and not, for example, `a*(b|cd)*`.

Optional extras

Because regex are really useful we end this tutorial with an optional introduction to regular expresssions. This does not depend on the technical content of the course, and will not be examined. It should give you a helpful introduction to the practical use of regex, and save you a lot of time in the long run.

In question 4 you will you will use regex for a task called *entity recognition*, to identify people, places, and things mentioned in Swift's *Gulliver's Travels*.

Gulliver's Travels

Here is a sample paragraph that mentions nine entities.

> Soon after my return from Leyden, I was recommended by my good master, Mr. Bates, to be surgeon to the Swallow, Captain Abraham Pannel, commander; with whom I continued three years and a half, making a voyage or two into the Levant, and some other parts. When I came back I resolved to settle in London; to which Mr. Bates, my master, encouraged me, and by him I was recommended to several patients. I took part of a small house in the Old Jewry; and being advised to alter my condition, I married Mrs. Mary Burton, second daughter to Mr. Edmund Burton, hosier, in Newgate-street, with whom I received four hundred pounds for a portion.

3 regex

A regular expression is a powerful way of specifying a pattern for a complex search. A regex is a search-string with wildcards—and more. It is a pattern that is matched against the text to be searched.

There are many different standards for expressing regular expressions, but there are some basic components shared by all standards. The general setting is that we have a finite alphabet of symbols, finite sequences of symbols are called strings.

We give rules for forming regular expressions, and say which strings they match.

- each symbol is a regular expression; it matches itself
- if two regex, R and S, that match strings r, s, then
  - RS is a regex matching r `++` s
  - $R|S$ is a regex thet matches both r and s
- if R is a regex then R^* is a regex; matches for R^* are given by two rules:
  - R^* matches the empty string `""`
  - if R^* matches s and R matches r then R^* matches s `++` r

The variables $R, S, \ldots$ range over regex; $r, s, \ldots$ to range over strings, and we set `strings in typewrite font`. We use parentheses to show the order in which a regex is built.

The common setting for regular expressions is text processing, where we take characters as symbols, but they have many applications, ranging from postcode checking to DNA searches.

* * * * * * *

To start, you will apply a regex to convert the format of a table as in this example:[7]

Original

```
Linda Fairstein (born 1947)
Anthony Faramus (1920-1990)
John Fante (1909-1983)
Nuruddin Farah (born 1945)
Nancy Farmer (born 1941)
Penelope Farmer (born 1939)
Philip Jos Farmer (born 1918)
Howard Fast (1914-2003)
```

[7] This transformation was accomplished by the regex shown above, with substitution `$8 $4, $1`.

```
Tarek Fatah (born 1949)
William Faulkner (1897-1962)
Madame de La Fayette (1634-1693)
```

Transformed

```
1947 Fairstein, Linda
1920 Faramus, Anthony
1909 Fante, John
1945 Farah, Nuruddin
1941 Farmer, Nancy
1939 Farmer, Penelope
1918 Farmer, Philip Jos
1914 Fast, Howard
1949 Fatah, Tarek
1897 Faulkner, William
1634 de La Fayette, Madame
```

```
^((([A-Z][a-z]+) )+)(([a-z]+ ([A-Za-z]* )*)?[A-Z][a-z]*) \((born )?([0-9]{4})([0-9]{4})?\)$
```

In practice, for two reasons, the syntax used for regex applications is more complicated than that suggested by the basic rules. First, it is useful to have shortcuts for common patterns, such as `a|b|c| ... | x | y | z` (commonly abbreviated as `[a-z]` or `[[:lower:]]`). Second we often want to treat *all* characters as symbols, but we need some characters to have special regex meanings, such as `|`, `*`, `( )`, and the characters we use to write short-cuts, which include `[ ] : - ^`. There are several incompatible standards that address these two issues. In this note we use the POSIX extended standard.

1. Find a regex tutorial on the web, or read the regex documentation for your favourite editor, and experiment with examples. But, for example, if you want to use regular expressions in EMACS you should look at this link (look also at the pages before and after the one linked)

2. Look at the list of writers on Wikipedia. Go to the list of *authors by name*, that includes surnames with the initial letter(s) that would match your name. Copy and paste your list of names from Wikipedia to a regex tester such as the one linked here. This is your text. This exercise asks you to try parts of the regex given above (select the flags `m, g`).[8] You will have to make some small alterations since the character used on the web page is different from the `-` used in this document. Use the regex tester to see what parts of the text are matched by various parts of the regex.

   - `([A-Z][a-z]+)`
   - `^((([A-Z][a-z]+) )+)`
   - `([A-Za-z]* )*)?[A-Z][a-z]*)`
   - `([0-9]4)`
   - `\((born )?([0-9]4)([0-9]4)?\)`
   - `([a-z]+ ([A-Za-z]* )*)`

   Experiment with regexes with groups to match the first names, the surname, and the birth year. Experiment with the substitutions. Adjust your regex so that it deals properly with your list of authors.

3. `CL2a.hs` includes declarations of `re::String` and `authors::String`

   The match operator:
   `(=~)::String -> String -> [[Str ing]]`
   comes from `Text.Regex.Posix`.

   The call, `authors =~ re ::[[String]]`
   returns a list of lists of strings.[9] For each match this lists the strings matched by the regex and each of its parenthesised subgroups.

   Enter your list of writers as a Haskell string and use Haskell to reformat it.

4. `CL2a.hs` also includes Gulliver's Travels, as a Haskell string. Its declaration begins like this: Look at the paragraph quoted on page 1. You will see nine proper names. In order of appearance, these are, Leyden, Mr. Bates, Swallow, Captain Abraham Pannel, Levant, London, Old Jewry, Mrs. Mary Burton, Mr. Edmund Burton, Newgate-street.

   (a) Give a regex that will match each of these, but nothing else in the paragraph.
   Hint: You want sequences of capitalised words. You don't want to put a capitalised word at the end of one sentence together with the first word of the next sentence, but you do want to keep Mr.

[8] Observe that in this regex, two of the parentheses are *escaped* `\( \)`. In this version of regex, ordinary parentheses are used to show the order in which a regex is built and form groups; an escaped parenthesis acts as a literal which matches a parenthesis in the target text. In the string literal for the regex in hte Haskell version in Question 3, the escape character `\` must itself be escaped, as `\\` .

[9] If we specify a `Boolean` result, `authors =~ re ::Bool` returns `True` iff there is a match.

and Mrs. together with their names. It may be helpful to observe that, in this text there are two spaces (or a newline) between one sentence and the next.

(b) Use regex in Haskell to produce a list of proper names occurring in the full text as you can. You should modify your regex if necessary to extract as many proper names as you can.

(c) Can you use regex to automatically classify your names into people, places and things?