Informatics 1 - CL: Tutorial 2

regex

Week 4: 8 – 12 October 2018

Please attempt the entire worksheet in advance of the tutorial, and bring all work with you. Tutorials cannot function properly unless you study the material in advance. Attendance at tutorials is **obligatory**; if you cannot attend your allocated tutorial go to another session. Please let the ITO know if you cannot attend.

You may work with others, indeed you should do so; but you must develop your own understanding; you can't phone a friend during the exam. If you do not master the coursework you are unlikely to pass the exams.

October 7, 2018

You can do part 1, on practical applications of regular expressions, at your leisure—you don't have to complete it before the tutorial, but we will allocate a few minutes for questions about this. We will find time to discuss your solutions later in the course— in the meantime you can discuss solutions on Piazza.

Sections 2 & 3 are technical.

I expect you to spend about one hour on section 2 (DFA-regex), and about two hours on section 3 (DFA in Haskell).

Tutorial activities to follow.

This tutorial exercise sheet was written by Matthew Hepburn, Dagmara Niklasiewicz, and Michael Fourman. Send comments to `Michael.Fourman@ed.ac.uk`

Introduction

In tutorial 1 we introduced finite state machines, and gave examples of deterministic transducers – machines with inputs and outputs. In this tutorial we will focus on simpler machines, the deterministic finite automata (DFA).

We have already identified the answers to yes/no, or `True/False`, questions as bits of information. A DFA gives us a `True/False` answer when presented with an input string. A deterministic finite state automaton (DFA) is like a transducer, but simpler, since it has no outputs.

We run the DFA just as we ran the transducers in tutorial 1, but instead of generating any outputs, we say the result of the computation – the answer to our `True/False` question – depends only on the final state. For some states the answer is `True`; for the rest it is `False`. The states for which the answer is `True` are called the *accepting* states.

DFA provide a simple model of computation. Each DFA 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, in due course, characterise exactly which questions about strings can be answered by a DFA. A regular expression (regex) is kind of pattern, It turns out that for each DFA 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 DFA.

Because regex are really useful we begin this tutorial with an 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.

1 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:[1]

Original	**Transformed**
`Linda Fairstein (born 1947)`	`1947 Fairstein, Linda`
`Anthony Faramus (1920-1990)`	`1920 Faramus, Anthony`
`John Fante (1909-1983)`	`1909 Fante, John`
`Nuruddin Farah (born 1945)`	`1945 Farah, Nuruddin`
`Nancy Farmer (born 1941)`	`1941 Farmer, Nancy`
`Penelope Farmer (born 1939)`	`1939 Farmer, Penelope`
`Philip José Farmer (born 1918)`	`1918 Farmer, Philip José`
`Howard Fast (1914-2003)`	`1914 Fast, Howard`
`Tarek Fatah (born 1949)`	`1949 Fatah, Tarek`
`William Faulkner (1897-1962)`	`1897 Faulkner, William`
`Madame de La Fayette (1634-1693)`	`1634 de La Fayette, Madame`

```
^((([A-Z][a-zé]+) )+)(([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,

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

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`).[2] 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-zé]+) )+)`
    - `([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`

```
re :: String
re = "^((([A-Z][a-z]+) )+)" ++


authors = "Linda Fairstein (born 1947)\n\
 \Anthony Faramus (1920-1990)\n\
 \John Fante (1909-1983)\n\
 \Nuruddin Farah (born 1945)\n\
 \Nancy Farmer (born 1941)\n\
 \Penelope Farmer (born 1939)\n\
 \Philip Jose Farmer (born 1918)\n\
 \Howard Fast (1914-2003)\n\
 \Tarek Fatah (born 1949)\n\
 \William Faulkner (1897-1962)\n\
```

The match operator:
`(=~)::String -> String -> [[String]]`
comes from `Text.Regex.Posix`.
The call, `authors =~ re ::[[String]]` returns a list of lists of strings.[3] 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.

[2] 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 `\\` .

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

4. CL2a.hs also includes Gulliver's Travels, as a Haskell string. Its declaration begins like this:

```
gulliverstravels :: String
gulliverstravels =         "GULLIVER'S TRAVELS\n\
\                                INTO SEVERAL\n\
\                        REMOTE NATIONS OF THE WORLD\n\
\ \n\
\ \n\
\                          BY JONATHAN SWIFT, D.D.,\n\
\                       DEAN OF ST. PATRICK'S, DUBLIN.\n\
```

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. 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.   This produces most of the names - but also allows many words that were capitalised at the start of a sentence.. Some regex dialects allow you to exclude some words from the match (so-called negative lookaround), but to do much better than this you would have to get your hands dirty.

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

* * * * * * *

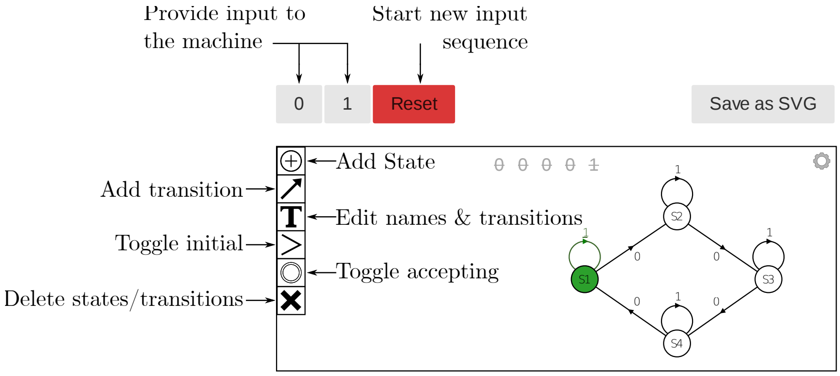
2 DFA-regex

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.

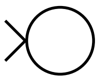
Using the FSM Workbench

The workbench provides tools for creating, editing, and simulating finite state machines. The diagram shows the function of each tool.

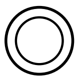
You can toggle each tool on/off by clicking it. When no tools are active you can drag the states of your FSM to rearrange the layout.



Like transducers, DFA are often represented as directed graphs. Nodes (circles) represent states. Accepting states are marked with an inner circle. The initial start state is indicated with an arrowhead.



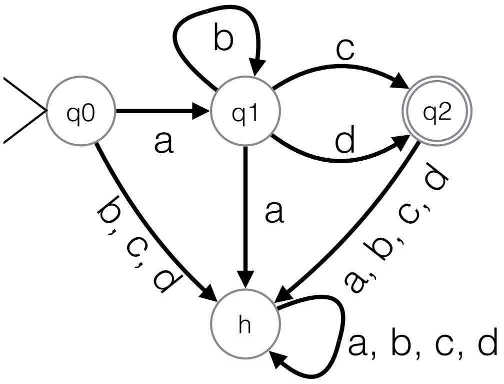
Initial



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.

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

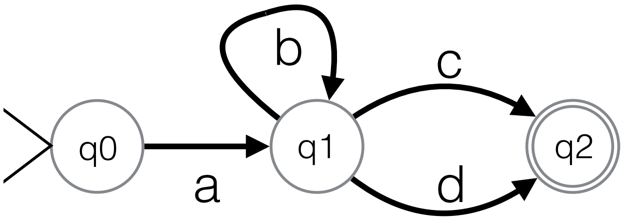
   (a) `"abbd"` (b) `"ad"` (c) `"aab"` (d) `"abbbc"` (e) `"ac"`

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).

In the 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*

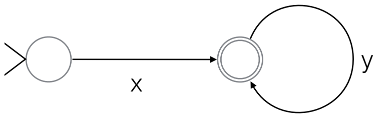


A very simple DFA

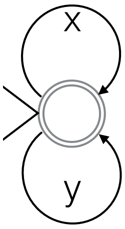
The following questions use the black hole convention - and you can use it in your answers.

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

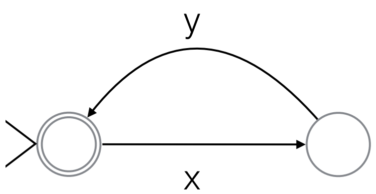
(a)



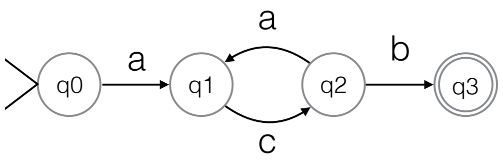
(b)



(c)



(d)



3. Draw a DFA that accepts the strings described by each of the following regex.[4]

   (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 a DFA may have any number of accepting states.

[4] 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)*`.

3 DFA in Haskell

We now use Haskell to implement a formal model of a DFA (the code is in `CL2.hs`).

A DFA is defined in MML (Ch. 17) as a tuple $(K, \Sigma, q_0, \delta, F)$ where K is a finite set of states; Σ is a finite set of symbols, the alphabet; $q_0 \in K$ is the initial state; $\delta : K \times \Sigma \longrightarrow K$ is the transition (next-state) function; $F \subseteq K$ is the set of accepting states.

We define a corresponding Haskell type parametrised by types, `state` and `symbol`.

```
type Next state symbol =
  state -> symbol -> state
type DFA state symbol = (
    [state]                 -- k
  , [symbol]                -- sigma
  ,  state                  -- q0
  , Next state symbol       -- delta
  , state -> Bool           -- f (accepting?)
  )
```

We introduce an abbreviation for the type of δ, the next state function (in curried form):

```
type Next state symbol = state -> symbol -> state
```

We represent the sets K and Σ as lists; q_0 is a state; a function `state -> Bool` tells us which states are accepting.

Here is the Haskell representation of our very simple DFA. The definition of `delta` has one line for each arrow of our diagram, followed by a catch-all that handles the black hole state.

```
simpleDFA :: DFA String Char
simpleDFA = (
    ["q0","q1","q2","h"]
  , "abcd"
  , "q0"
  , delta
  , f
  )
  where
    delta "q0" 'a' = "q1"
    delta "q1" 'b' = "q1"
    delta "q1" 'c' = "q2"
    delta "q1" 'd' = "q2"
    delta _    _   = "h"
    f "q2" = True
    f _    = False
```

The definition of `final` is just like the one we gave for transducers in tutorial 1, but we make the first parameter a DFA, from which we extract the transition function. The pattern `dfa@(_,_,_,delta,_)` (which we say as "`dfa as (_,_,_,delta,_)`") binds `dfa` to the DFA *and* `delta` to its transition function.

```
final :: DFA st sy ->  st -> [sy] -> st
final dfa                 s   []    = s
final dfa@(_,_,_,delta,_) s (x:xs) = final dfa (delta s x) xs
```

We use the same trick again to extract the start state, `q0`, and the acceptance predicate, `f`, and call `final` to define `accept`.

```
accept :: DFA state symbol -> [symbol] -> Bool
accept dfa@(_, _, q0, _, f) xs = f (final dfa q0 xs)
```

4. (a) Use `accept` to check your results for question 1.

   (b) Represent each DFA from question 2 as a Haskell value of type `DFA String Char`.

5. The function basebmodm was introduced in lecture 5.

   (a) Draw the DFA `base3mod2 = basebmodm 2 3` which accepts strings that are ternary representations of even numbers.

   (b) Give a regular expression for this set of strings.

   (c) How can you easily check whether a number written in ternary notation is odd or even?

```
basebmodm :: Int -> Int
  -> DFA Int Int
basebmodm m b = (
    [0..(m-1)]  -- states
  , [0..(b-1)]  -- symbols
  , 0           -- start at 0
  , delta       -- see below
  , (==0)       -- accepts 0
  ) where
  delta st sy =
    (b*st + sy) `mod` m
```

5. The product of two DFA, which accepts the intersection of the languages accepted by each machine, is given by the following definition:

```
prodDFA :: Eq sy => DFA st sy -> DFA st' sy -> DFA (st, st') sy
prodDFA (k,  sigma,  q0,  delta,  f)
        (k', sigma', q0', delta', f') =
  if sigma /= sigma' then undefined else
    (
      [(x,x') | x <- k, x' <- k']
    , sigma
    , (q0, q0')
    , delta''
    , f''
    )where
      delta'' (x, x') s = (delta x s, delta' x' s)
      f'' (q, q') = (f q) && (f' q')
```

   (a) The sum of two DFA accepts the union of the languages accepted by each machine. How would you modify the code to produce the sum DFA?

   (b) Draw the product and sum of the two DFA , 2a, 2c, from question 2. Give regular expressions for the languages they accept.

6. (a) Write a Haskell function `nary :: Int -> Int -> [Int]` such that, for positive n $>$ 1 and non-negative `x`, `nary n x` represents `x` in `n`-nary notation, as a list of numbers less than `n` –with the most significant digit at the head of the list. For `n` $=$ 1, `nary 1 x` should return the unary representation of the non-negative number `x`.

   (b) Write a function `dest :: DFA st sy -> [sy] -> st` such that `dest dfa xs` computes the destination state reached by the machine `dfa` given the inputs `xs`. *Hint:* similar to `accept` — call `final` with the the machine's start state.

   (c) Define `prop`, as below, then call `quickCheck prop` (or `verboseCheck prop`).

```
prop :: (Positive Int) (Positive Int) (NonNegative Int) -> Bool
prop (Positive b) (Positive m) (NonNegative x) =
   dest (basebmodm m b) (nary b x) == x `mod` m
```

   This checks, for positive m and b, and non-negative x, that the machine `basebmodm m b`, given the b-ary representation of x as input, reaches the state $(x \bmod n)$.

Tutorial Activity

regex

- As announced, the practical regex exercise will not be covered in tutorials. This material will not be examined, but we hope you have learnt some useful skills. We welcome your feedback on this task.

DFA-regex

10m Start by going through Questions 1–3. Work in your groups to agree on your answers.

10m We will go through the answers to questions 1 and 2, Then ask groups to contribute different (correct) answers to question 3.

20m Consider the regex introduced in question 3, as five pairs of regex. For each pair, ab, cd, ef, gh, ij, you should have two machines.

Work as a group, dividing the task amongst you however you choose, to produce a product machine, and the corresponding regex for the intersection of the two languages, for each pair. You only need to include the states *reachable* from the start state of the product machine – those states reached as the destination state for some input sequence.

DFA in Haskell

10m Check your answers to questions 4 and 5 with your buddy.

10m Move on to question 5. You should make sure you understand the construction of the product and sum machines, and the Haskell implementation of these operation,

30m Check your answers for question 6. Make sure you understand the Haskell code for the test property, and what the property tells us about the operation of the DFAs we can construct with the `basebmodm` function.

Working as a group, use these ideas to check (using quickCheck) whether the product of the two machines `base2mod3` and `base2mod2` recognises the same language as `base2mod6`.

This tutorial exercise sheet was written by Matthew Hepburn and Dagmara Niklasiewicz, with additions from Michael Fourman. Send comments to `Michael.Fourman@ed.ac.uk`