Informatics 1 - Computation & Logic: Tutorial 7

Computation: Non-Deterministic FSMs and Regular Expressions

Week 9: 13 - 17 November 2017

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**; please let the ITO know if you cannot attend.

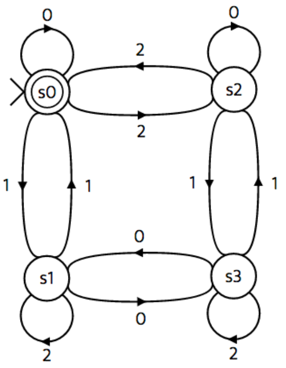
You may work with others, indeed you should do so; but you must develop your own understanding; you can't phone a friend during the exam. If you do not master the coursework you are unlikely to pass the exams.

You may find it useful to refer to the FSM Workbench question set which accompanies this tutorial at homepages.inf.ed.ac.uk/s1020995/tutorial6.

Use the workbench to check your working, but ultimately you should aim to be able to answer simple questions of this type working with just pencil and paper.

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`

In Lecture 14 we introduced this example of a DFA encoded in Haskell:

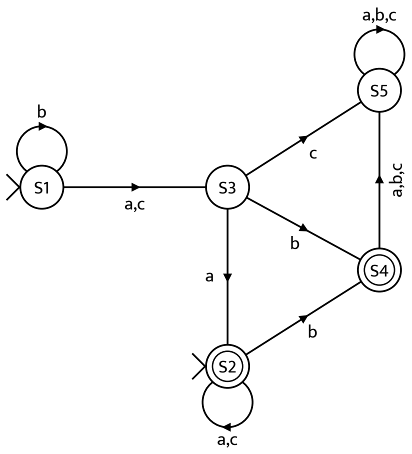


```
      s0 []   =  True              s2 []   =  False
s0 ('0':xs)   =  s0 xs       s2 ('0':xs)   =  s2 xs
s0 ('1':xs)   =  s1 xs       s2 ('1':xs)   =  s3 xs
s0 ('2':xs)   =  s2 xs       s2 ('2':xs)   =  s0 xs

      s1 []   =  False             s3 []   =  False
s1 ('0':xs)   =  s3 xs       s3 ('0':xs)   =  s1 xs
s1 ('1':xs)   =  s0 xs       s3 ('1':xs)   =  s2 xs
s1 ('2':xs)   =  s1 xs       s3 ('2':xs)   =  s3 xs
```

The code consists of four mutually recursive functions, one for each state of the machine. To check whether a string, `input`, is accepted by the machine we evaluate `s0 input` (because $S0$ is the starting state of our DFA), which returns the Boolean answer, `True` or `False`.
Check that you understand how this code relates to the DFA. How would you modify the code if the accepting state were $S3$? How could you modify the code to return the number of the final state of the trace generated by any input string?

1. Consider the finite state machine in the diagram below.



Input	Is Accepted?
⟨ ⟩	Y
b	Y
aa	Y
ba	N
abaab	N
acaca	Y
aaab	Y
bbbcb	Y
cacba	N

   (a) For each input sequence in the table above, record whether it is accepted by the FSM.

   (b) Is the FSM deterministic? Justify your answer.
   NO. As it has two initial states it can be in more than one state for some input sequences.

   (c) Implement a Haskell function to check your answers (by implmenting a suitable DFA). The state transition table for the DFA is given by the subset construction:

	a	b	c
s1, s2	s2, s3	s1, s4	s2, s3
s2, s3	s2	s4	s2, s5
s1, s4	s3, s5	s1, s5	s3, s5
s2	s2	s4	s2
s4	s5	s5	s5
s5	s5	s5	s5
s2, s5	s2, s5	s4, s5	s2, s5
s3, s5	s2, s5	s4, s5	s5
s1, s5	s3, s5	s1, s5	s3, s5
s4, s5	s5	s5	s5

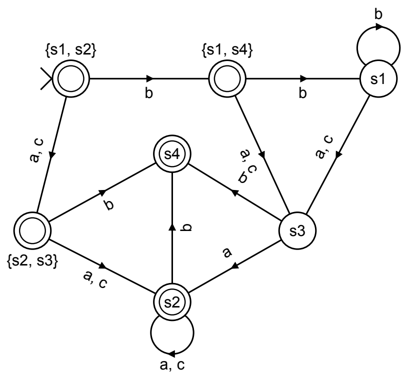
Discussion point: note that `s5` is a black hole state. This means that the oroginal machine is equivalent to the NFA given by deleting `s5` and all transitions to it. Working from this NFA gives us a simpler, but equivalent DFA.

	a	b	c
s1, s2	s2, s3	s1, s4	s2, s3
s2, s3	s2	s4	s2
s1, s4	s3	s1	s3
s2	s2	s4	s2
s4	{}	{}	{}
{}	{}	{}	{}
s3	s2	s4	{}
s1	s3	s1	s3

Haskell functions corresponding to these two DFA are given below

```
s1s2 (x:xs) = case x of
               'a' -> s2s3 xs
               'b' -> s1s4 xs
               'c' -> s2s3 xs
s1s2 [] = True
s2s3 (x:xs) = case x of
               'a' -> s2 xs
               'b' -> s4 xs
               'c' -> s2s5 xs
s2s3 [] = True
s1s4 (x:xs) = case x of
               'a' -> s3s5 xs
               'b' -> s1s5 xs
               'c' -> s3s5 xs
s1s4 [] = True
s2 (x:xs) = case x of
               'a' -> s2 xs
               'b' -> s4 xs
               'c' -> s2 xs
s2 [] = True
s4 (x:xs) = case x of
               'a' -> s5 xs
               'b' -> s5 xs
               'c' -> s5 xs
s4 [] = True
s5 (x:xs) = case x of
               'a' -> s5 xs
               'b' -> s5 xs
               'c' -> s5 xs
s5 [] = False
s2s5 (x:xs) = case x of
               'a' -> s2s5 xs
               'b' -> s4s5 xs
               'c' -> s2s5 xs
s2s5 [] = True
s3s5 (x:xs) = case x of
               'a' -> s2s5 xs
               'b' -> s4s5 xs
               'c' -> s5 xs
s3s5 [] = False
s1s5 (x:xs) = case x of
               'a' -> s3s5 xs
               'b' -> s1s5 xs
               'c' -> s3s5 xs
s1s5 [] = False
s4s5 (x:xs) = case x of
               'a' -> s5 xs
               'b' -> s5 xs
               'c' -> s5 xs
s4s5 [] = True
```

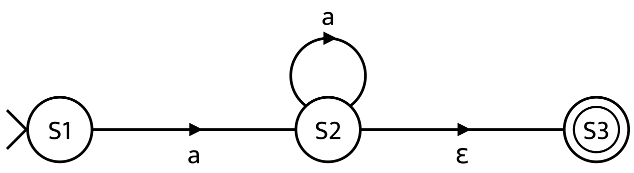
```
s1s2 (x:xs) = case x of
               'a' -> s2s3 xs
               'b' -> s1s4 xs
               'c' -> s2s3 xs
s1s2 [] = True
s2s3 (x:xs) = case x of
               'a' -> s2 xs
               'b' -> s4 xs
               'c' -> s2 xs
s2s3 [] = False
s1s4 (x:xs) = case x of
               'a' -> s3 xs
               'b' -> s1 xs
               'c' -> s3 xs
s1s4 [] = True
s2 (x:xs) = case x of
               'a' -> s2 xs
               'b' -> s4 xs
               'c' -> s2 xs
s2 [] = True
s4 (x:xs) = False
s4 [] = True
s3 (x:xs) = case x of
               'a' -> s2 xs
               'b' -> s4 xs
               'c' ->  xs
s3 [] = False
s1 (x:xs) = case x of
               'a' -> s3 xs
               'b' -> s1 xs
               'c' -> s3 xs
s1 [] = False
```



Note that in the second case we don't implement a function corresponding to the black hole state - instead, we immediately return `False` whenever we would otherwise call that function.

There is still one opportunity to optimise this machine. Use the FSM workbench to find a minimal DFA for the machine, and convince yourself that it recognises the same language. You will learn more about minimal DFA next year.

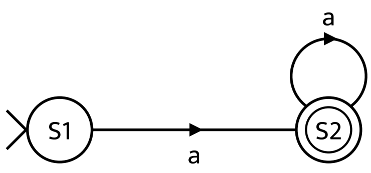
2. This NFA over the alphabet $\{a\}$ uses an ε transition.



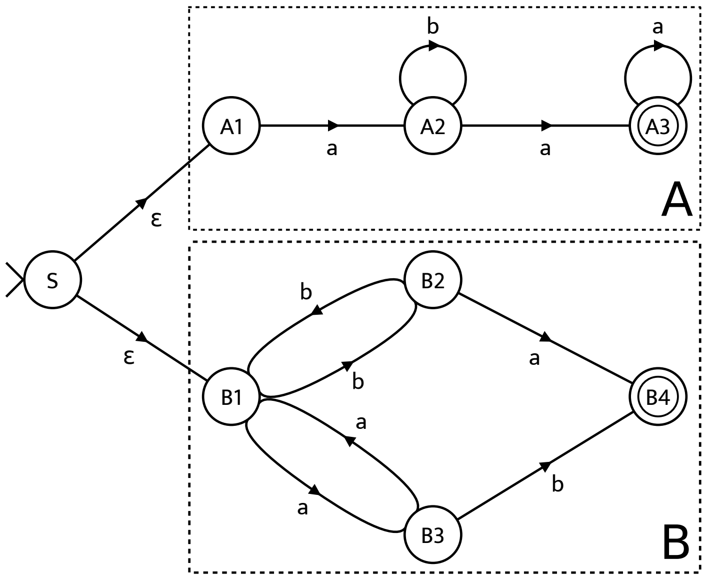
   (a) Describe the language accepted by this machine in words.
   All sequences of ‘a’ that are at least one character long

   (b) Describe the language accepted by this machine using a regular expression. aa*

   (c) Design a deterministic machine that accepts the same language as this machine.



3. ε-transitions provide a simple way of combining FSMs. The machine below has been composed from two machines A and B, which had initial states A1 and B1.



(a) Considering machines A and B separately, give a regular expression which describes the language they accept. A: `ab`*`aa`*
B: (`bb`|`aa`)*(`ba`|`ab`)

(b) Considering the whole machine, give a regular expression which describes the language the machine accepts.
(`ab`*`aa`*)|((`bb`|`aa`)*(`ba`|`ab`))

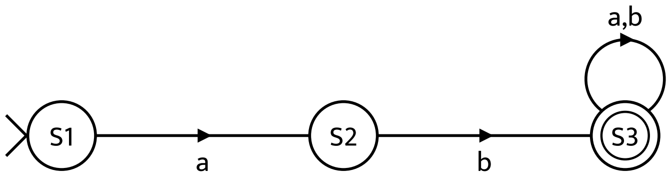
(c) L_A and L_B are the languages accepted by machines A and B. Give an expression relating L_A and L_B to L, where L is the set of input accepted by the whole machine. $L = L_A \cup L_B$

(d) Construct and test a Haskell implementation of an equivalent DFA.

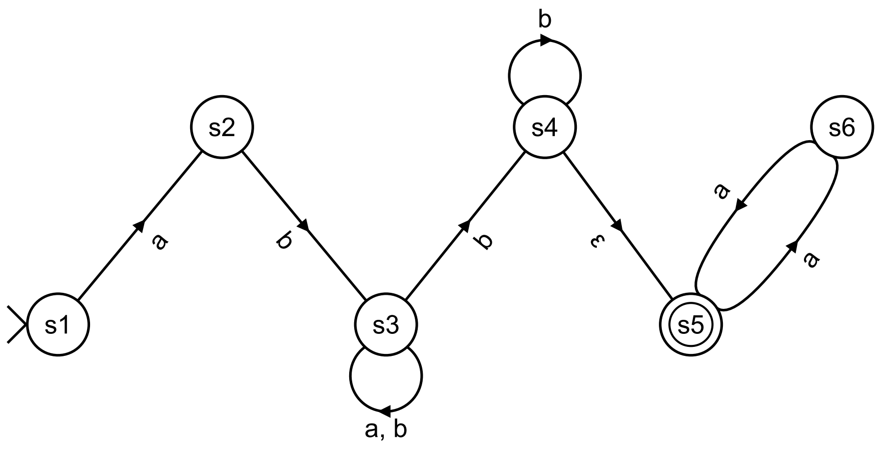
4. Consider the regular expression `ab`(`a`|`b`)*

(a) Describe in words the language that the expression matches. Include two examples of strings that are matched. The string '`ab`' followed by zero or more '`a`'s and '`b`'s. Examples of accepted strings include '`ab`', '`abaaa`', and '`ababba`'.

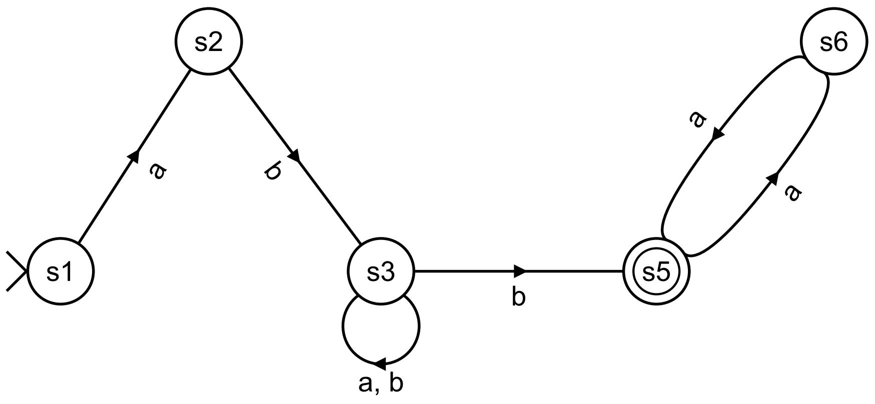
(b) Design a finite state machine that accepts that language.



(c) Building on your answer to (b), design a finite state machine that accepts `ab`(`a`|`b`)*`bb`*(`aa`)*.

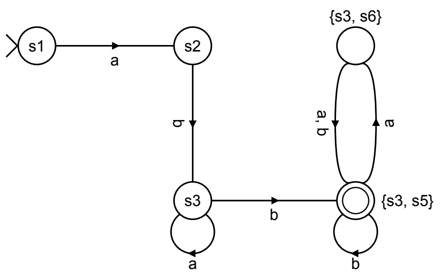


Is the obvious answer, but this is equivalent to



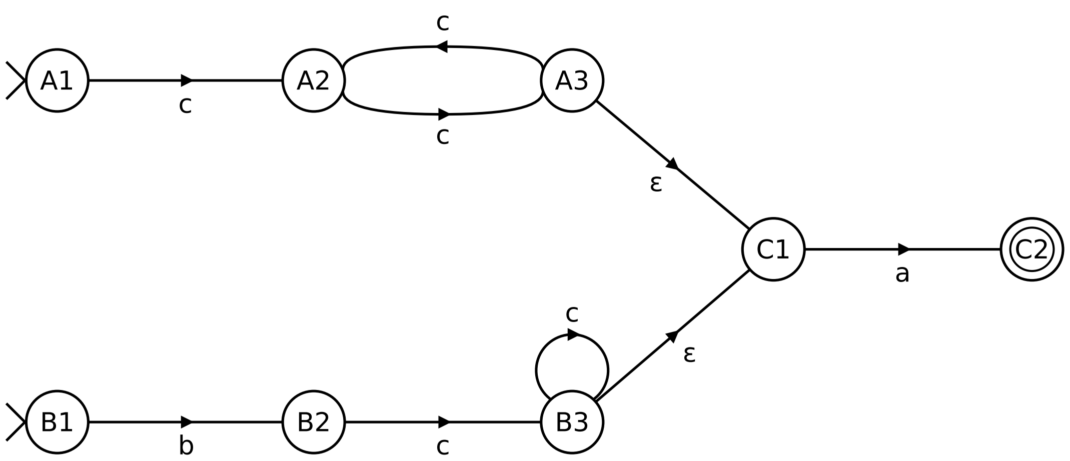
We can give this simpler FSM since, $(\mathtt{a}|\mathtt{b})^*\mathtt{bb}^* \equiv (\mathtt{a}|\mathtt{b})^*\mathtt{b}$, which follows from the facts that $\mathtt{bb}^* \equiv \mathtt{b}^*\mathtt{b}$ and $(\mathtt{a}|\mathtt{b})^*\mathtt{b}^* \equiv (\mathtt{a}|\mathtt{b})^*$.

(d) Construct and test a Haskell implementation of an equivalent DFA.



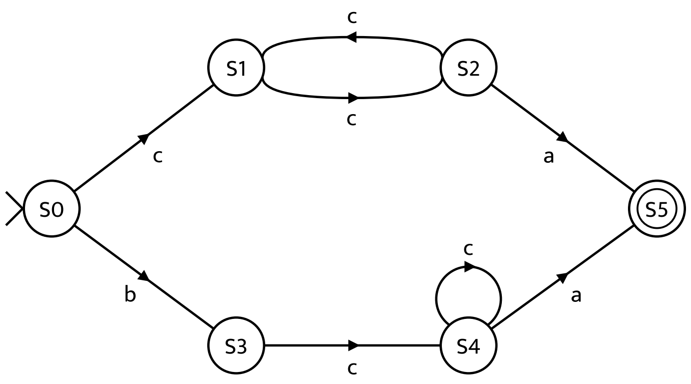
```
s1 ('a':xs) = s2 xs
s1 _ = False
s2 ('b':xs) = s3 xs
s2 _ = False
s3 ('a':xs) = s3 xs
s3 ('b':xs) = s3s5 xs
s3 _ = False
s3s5 ('a':xs) = s3s6 xs
s3s5 ('b':xs) = s3s5 xs
s3s5 [] = True
s3s6 (x: xs) = s3s5 xs
s3s6 [] = False
```

5. Consider this NFA over the alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.



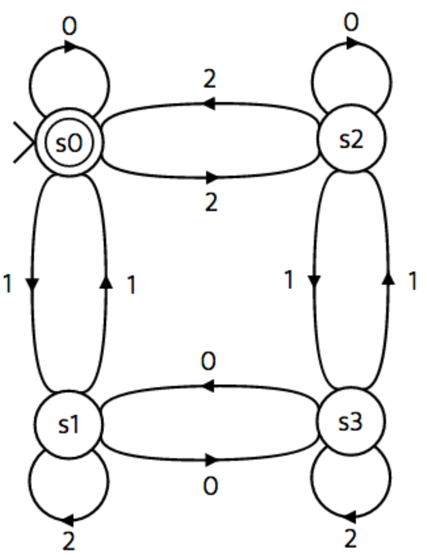
(a) Describe, both in words and with a regular expression, the language accepted by this machine. Hint: think about the sequences that end in A3 and B3. The machine accepts strings consisting of an even number (>0) of `c`'s followed `a` or `b` followed by at least one `c` followed by `a`.
(c(cc)*ca)|(bcc*a)

(b) Design a DFA that accepts the same language.



(c) Are there any NFAs that cannot be converted into an equivalent DFA? No – all NFAs have an equivalent DFA.

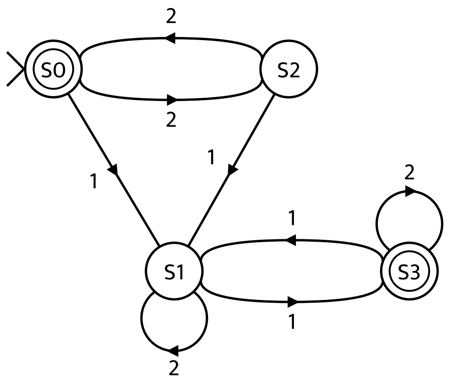
6. Consider this DFA over the alphabet $\{0, 1, 2\}$. It should be familiar.



(a) Describe, in words, the language accepted by this machine. Hint: Your description in words should refer to ternary numbers. This machine accepts ternary representations of natural numbers divisible by 4.

(b) Replace each transition labelled 0 by a transition labelled ε, between the same two states. The resulting automaton is not a DFA. (Why not?)
Because a DFA has no ε transitions.

i. Construct an equivalent DFA.

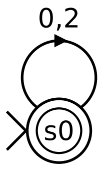


ii. Describe, both in words and with a regular expression, the language accepted by this machine.

There are lots of equivalent answers you can derive for this one – but seeing that they are equivalent is not always easy. You can do this using Arden's rule, but it is fairly easy to see that a string on the alphabet $\{1, 2\}$ that is accepted has an even number of 1's, and if it has no 1's then it has an even number of 2's : $(22)^*(2^*12^*12)^*$. (L_{S1} is the set of strings with an odd number of ones.) For any string with a positive even number of ones, or no ones and an even number of twos, we can find easily an accepting trace, so this description characterises the accepting strings. A more-pleasing regular expression is $(22)^*(2^*12^*12^*)^*$, which is equivalent to $(2^*12^*12^*)^*(22)^*$.

(c) Next, replace each transition (of the original machine) labelled 1 by a transition labelled ε, between the same two states.

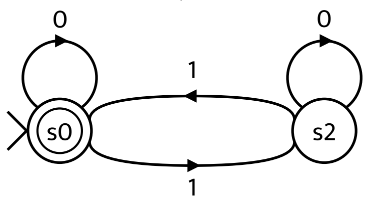
i. Again, construct an equivalent DFA, and, ii, describe the language it accepts.



Any string on the alphabet $\{0, 2\}$: $(2 \mid 0)^*$

(d) Repeat the exercise replacing each transition (of the original machine) labelled 2 by a transition labelled ε, between the same two states.

i. Construct an equivalent DFA,



ii. describe the language it accepts.

Any string on the alphabet $\{0, 1\}$ with an even number of 1's : $(0^*10^*10^*)^*0^*$

(e) BONUS QUESTION: Give a regular expression that describes the language accepted by the original machine. Test your answer using the grep utility.

This bonus question goes somewhat beyond the call of duty. Feel free not to attempt it. That said, by the end of week 7 you should have all the tools required to complete it. If you do choose to try it, I suggest you use cut and paste in some suitable editor to make, and keep track of the algebraic substitutions that are required.

By repeated application of Arden's Lemma, substitution and simplification, from the equations

$$\begin{aligned}
L_0 &= L_0 0 \mid L_1 1 \mid L_2 2 \mid \varepsilon \\
L_1 &= L_0 1 \mid L_1 2 \mid L_3 0 \\
L_2 &= L_0 2 \mid L_2 0 \mid L_3 1 \\
L_3 &= L_1 0 \mid L_2 1 \mid L_3 2
\end{aligned}$$

For example, applying Arden's Lemma to the final equation for L_3 and then substituting the result for L_3 in the equations for L_1 and L_2, we obtain,

$$\begin{aligned}
L_0 &= L_0 0 \mid L_1 1 \mid L_2 2 \mid \varepsilon \\
L_1 &= L_0 1 \mid L_1 2 \mid (L_1 0 \mid L_2 1) 2^* 0 \\
L_2 &= L_0 2 \mid L_2 0 \mid (L_1 0 \mid L_2 1) 2^* 1 \\
L_3 &= (L_1 0 \mid L_2 1) 2^*
\end{aligned}$$

We can apply distributivity $(x \mid y)z = (xz \mid yz)$, for regular expressions, together with the commutativity and associativity of $|$, to regroup the equation for L_2 to a form suitable for Arden.

$$\begin{aligned}
L_2 &= L_0 2 \mid L_2 0 \mid (L_1 0 \mid L_2 1) 2^* 1 \\
&= L_0 2 \mid L_2 0 \mid L_1 0 2^* 1 \mid L_2 1 2^* 1 && \text{(distrib)} \\
&= L_0 2 \mid L_1 0 2^* 1 \mid L_2 0 \mid L_2 1 2^* 1 && \text{(comm)} \\
&= L_0 2 \mid L_1 0 2^* 1 \mid L_2 (0 \mid 1 2^* 1) && \text{(distrib)} \\
L_2 &= (L_0 2 \mid L_1 0 2^* 1)(0 \mid 1 2^* 1)^* && \text{(Arden)}
\end{aligned}$$

Continuing in similar vein, to eliminate L_2 and L_1, we eventually apply Arden to an equation for L_0, to derive
$L_0 = \texttt{(0|2(0|12}^*\texttt{1)}^*\texttt{2|(1|2(0|12}^*\texttt{1)}^*\texttt{12}^*\texttt{0)(2|02}^*\texttt{0|02}^*\texttt{1(0|12}^*\texttt{1)}^*\texttt{12}^*\texttt{0)}^*\texttt{(1|02}^*\texttt{1(0|12}^*\texttt{1)}^*\texttt{2))}^*$

7. Use the FSM workbench to construct a machine that accepts ternary strings with an even number of 1s and an even number of 2s (and any number of 0s) that represent a number that is not a multiple of four.

   Hint: *start by constructing a machine that accepts ternary strings S that satisfy at least one of the following three conditions:*

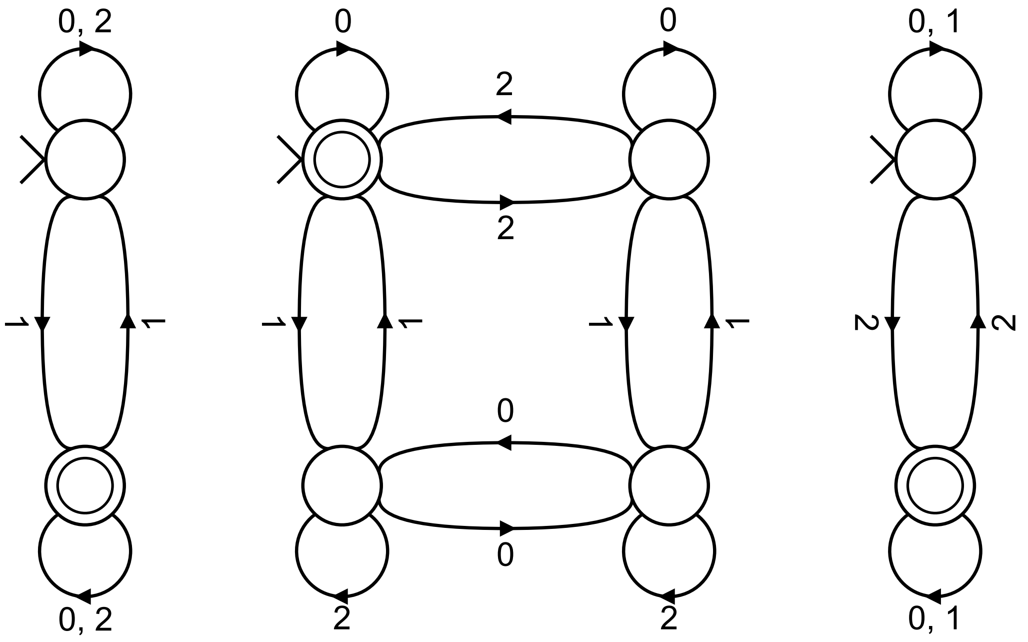
   (a) S includes an odd number of 1*s,*

   (b) S includes an odd number of 2*s,*

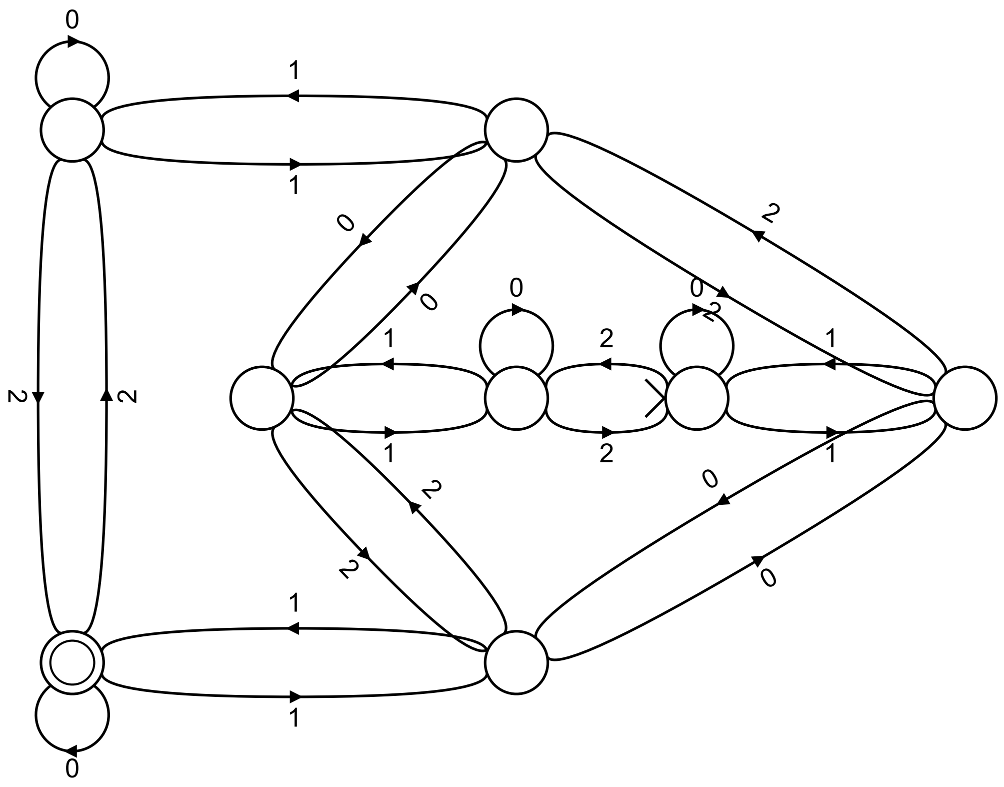
   (c) S represents a multiple of 4*.*

   You are asked to build a machine that accepts strings that satisy none of these conditions.

Following the hint, construct the machine



Use the workbench to convert this to DFA, then change accepting states to give the complement.



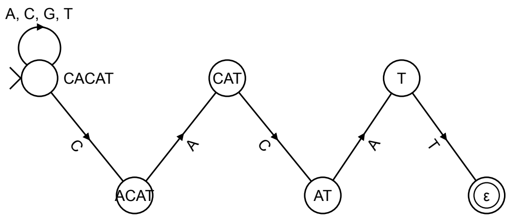
Tutorial Activity

String matching algorithms try to find a place where one or several strings (also called patterns) are found within a larger string or text.
String matching algorithms play a vital role in a host of applications ranging, for example, from the detection of plagiarism, to the analysis of protein and DNA sequences in Computational Biology.
In computational biology DNA – the stuff of which the double helixes that carry genetic information are made – consists of two chains of bases.[1] There are four types of base: cytosine (C), guanine (G), adenine (A) or thymine (T). The two chains are matched together A–T and C–G, so one chain determines the other. For example, if one chain is AATCAG the other must be TTAGTC.

[1]Biology is infinitely complex. This is a simplified account for the purposes of this exercise.

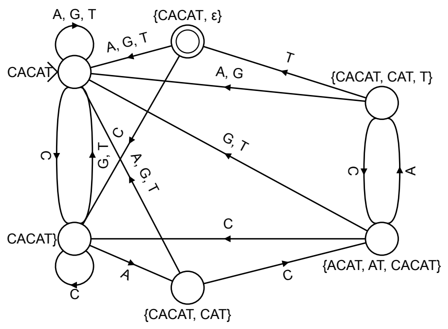
In real-life problems biologists may search for patterns with thousands of letters in genomes with billions of base-pairs.
For this exercise you should consider strings on the alphabet with four symbols AGCT.

1. (a) Draw an NFA that will accept any string that ends with the pattern CACAT. Name each state with the string it is looking for – a string that has a trace from that state to an accepting state. So, the starting state will have the name CACAT and the accepting state has the name ϵ.

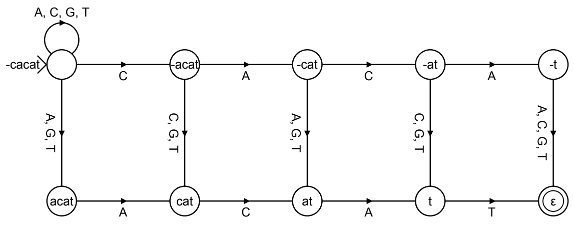


   (b) List the reachable states of the NFA, as constructed by the subset procedure.

- CACAT
- ACAT, CACAT
- CAT, CACAT
- AT, ACAT, CACAT
- T, CAT, CACAT



2. (a) Draw an NFA that will accept any string that includes a close match to the pattern CACAT, where a close match is either exactly this string, or a string CACATof the same length that differs from the given pattern at at most one letter. Hint: some states will be looking for an exact match, and some for a close match.



The top row includes states looking for a string that may include an error; the bottom row is reached when we encounter an error – so we must find an exact match for the remainder of the search string.

3. If time permits, derive a Haskell program inplementing the DFA for the first of these tasks. Each state of the DFA corresponds to a set of states of the NFA – each state of the DFA is looking for any one of the strings corresponding to the NFA states it includes. If we fail to find the first letter of the shortest string we are looking for, we can fall back to look for the next shortest string.

```
cacat ('c':xs) = acat xs
cacat (x:xs) = cacat xs
cacat [] = False
acat ('a':xs) = cat xs
acat ('c':xs) = acat xs
acat (x:xs) = cacat xs
acat [] = False
cat ('c':xs) = at xs
cat ('x':xs) = cacat xs
cat [] = False
at ('a':xs) = t xs
at ('c':xs) = acat xs
at (_:xs) = cacat xs
at [] = False
t ('t':xs) = True
t ('c':xs) = at xs
t [] = False
```

```
cacat ('c':xs) = acat xs
cacat (x:xs) = cacat xs
cacat [] = False
acat ('a':xs) = cat xs
acat xs = cacat xs
cat ('c':xs) = at xs
cat xs = cacat xs
at ('a':xs) = t xs
at xs = acat xs
t ('t':xs) = True
t xs = cat xs
```

We name each state by the shortest string it is searching for. Here are two equivalent implementations. The first directly translates the DFA.
The second implementation makes use of a special property of simple string search. For example, if we reach the state `CACAT, CAT, T`, we have already seen the string `CACA`, and our search could be completed by finding any one of the three strings `CACAT, CAT, T`. We examine the next letter of the input, call it x. If it matches the first letter of the shortest string we move on. If it fails this match, we try the next shortest. The direct implementation of the DFA precomputes the next move, based on the value of x. In this example, if x is `C`, we call the function for state `ACAT, AT, CACAT` with the remaining input.
We can instead simply pass the problem including the letter we have just examined back to the function we have already written for the state `CACAT, CAT`. We do this in the second implementation. This is certainly easier to write, and it may be faster than the DFA implementation.
This second implementation corresponds to the famous KMP algorithm:
https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm