2014 Haskell January Test
Regular Expressions and Finite Automata

This test comprises four parts and the maximum mark is 25. Parts I, II and III are worth 23 of the 25 marks available. The **2014 Haskell Programming Prize** will be awarded for the best attempt(s) at Part IV.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You may therefore wish to define your own tests to complement the ones provided.

1 Introduction

Regular expressions, or *regexes*, are used to describe search patterns in text processing applications. Examples of their use include Linux's `grep` filter utility, text editors like `vim` and `emacs` and also programming languages like Ruby, Java and, of course, Haskell! This exercise aims to explore some of the algorithms and data structures traditionally associated with regular expression implementations.

2 Regular Expressions

For the purposes of this exercise a regular expression is defined recursively to be either:

1. A *terminal* character that defines a pattern that matches only that character in a given input string. For example the expression a defines a pattern that matches only the string `"a"`.

2. A *sequence* of two juxtaposed regular expressions. For example, the regular expression ab comprises two terminals, a and b, in sequence and this defines a pattern that will match only the string `"ab"`. Note that parentheses can optionally be used to bracket subterms of a regular expression in the usual way, and that, for example, $a(bc)$, $(ab)c$ and abc all represent the same expression (i.e. sequencing is associative).

3. An *alternative*, written $(e \mid e')$ where e and e' are regular expressions. For example, the expression $(ab \mid c)$ defines a pattern that matches *either* the string `"ab"` *or* the string `"c"`. Note that the $\mid$ operator is associative and commutative, so the expressions $(a \mid bb \mid ac)$, $(ac \mid (bb \mid a))$ and $((a \mid ac) \mid bb)$ all define the same matching pattern. In this exercise we will always enclose alternative expressions in parentheses.

4. A *repetition* ("Kleene star"[1]) of *zero or more* occurrences of a given expression, e say, written e^*. For example, ab^*c defines a pattern that matches the character `'a'` followed by zero or more occurrences of the character `'b'` followed by a single occurrence of the character `'c'`.

5. A repetition (Kleene plus) of *one or more* occurrences of a given expression, e, written e^+. For example, ab^+c defines a pattern that matches the string `"abbbc"`, but not `"ac"`, as there must be at least one `'b'` following the initial `'a'`.

6. An *optional* occurrence of a given expression, e, written $e?$. For example, $(ab)?d^+$ defines a pattern that matches the strings `"d"`, `"dd"`, `"abd"`, `"abdd"` and so on.

7. The *null* expression, which matches only the empty string (`""`). For example, the expression $(a \mid)$ defines a pattern that matches the strings `"a"` and `""` only. A common convention is to denote null expressions explicitly by the special symbol ϵ, as in $(a \mid \epsilon)$ for example, and this idea will feature later on in Section 3.

Some more examples of regular expressions and matching/non-matching strings are shown in Table 1. If a regular expression e defines a pattern that matches a string `s` then we say that e *accepts* `s`. For example, the expression $(ab \mid c)^*$ accepts the strings `""`, `"c"`, `"abababccab"` etc., as shown in the table.

[1] Named after the American mathematician Stephen Cole Kleene.

Regex	Matching string examples	Non-matching string examples
$(x \mid y)(1 \mid 2)$	`"x2"` `"y1"`	`"x"` `"x3"`

x'^*	`"x"`
	`"x'"`
	`"x'''"`
		`"y"`
		`"'"`
		`"x'x"`

$(ab \mid c)^*$	`""`
	`"c"`
	`"abababccab"`
		`"d"`
		`"ac"`
		`"cccb"`

$(a \mid)a$	`"a"` `"aa"`	`"b"` `"aaa"`

$(ab)?d^+$	`"d"`
	`"abd"`
	`"abddd"`
		`"ab"`
		`"bd"`
		`"ababd"`

Table 1: Some regular expressions and examples of matching/non-matching strings

2.1 Simplification Rules

The $^+$ and ? operators are, in fact, unnecessary and can expressed in terms of the other operators via the identities:

$$\begin{aligned} e^+ &\equiv ee^* \\ e? &\equiv (e \mid) \end{aligned}$$

for all expressions e.

2.2 Representation

Regular expressions, as described above, can be implemented in Haskell via the following data type:

```
data RE = Null      |       -- Null expression
          Term Char |       -- Terminal
          Seq RE RE |       -- Sequence
          Alt RE RE |       -- Alternation
          Rep RE    |       -- Repetition (*)
          Plus RE   |       -- Repitition (+)
          Opt RE    |       -- Optional expression (?)
        deriving (Eq, Show)
```

Table 2 gives some examples of regular expressions whose representations are defined in the template file for testing purposes.

You are now in a position to answer the questions in Part I. You may wish to complete these before reading on.

Regex	Representation	Template variable
$(x \mid y)(1 \mid 2)$	Seq (Alt (Term 'x') (Term 'y')) (Alt (Term '1') (Term '2'))	`re1`
x'^*	Seq (Term 'x') (Rep (Term '''))	`re2`
$(ab \mid c)^*$	Rep (Alt (Seq (Term 'a') (Term 'b')) (Term 'c'))	`re3`
$(a \mid)a$	Seq (Alt (Term 'a') Null) (Term 'a')	`re4`
$(ab)?d^+$	Seq (Opt (Seq (Term 'a') (Term 'b'))) (Plus (Term 'd'))	`re5`

Table 2: Example representations defined in the template

3 Non-deterministic Automata

A key property of regular expressions is that the patterns they define can be described by a *Non-Deterministic (Finite) Automaton*, or NDA, which is an example of a labelled state transition system with a finite number of states. An example NDA for the regular expression $(a \mid b)^*c$ (`reFigure` in the template) is shown in Figure 1.

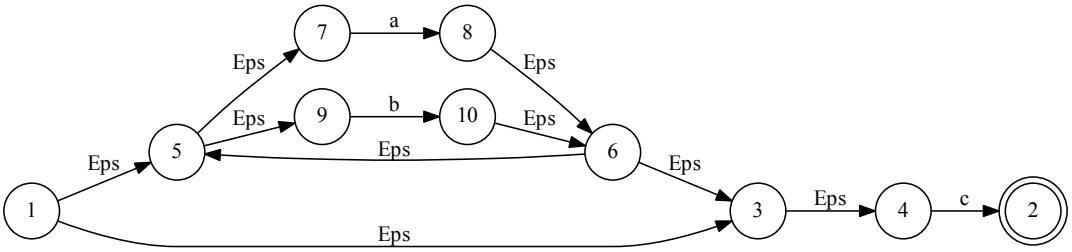


Figure 1: NDA for $(a \mid b)^*c$

An NDA has a unique *start* state (state 1 in Figure 1) and a unique *accepting* state, which we will also refer to as the *terminal* state, which is shown as a double circle (state 2 in Figure 1). The sequence of non-`Eps` transition labels visited on a path through the NDA from the start state to the accepting state defines a string that is "accepted" by, i.e. can be matched by, the regular expression. There may be many such paths, indeed possibly infinitely many, and these collectively determine the set of strings that are accepted by the regular expression.

A transition labelled with the special symbol `Eps` corresponds to a null regular expression[2] and can be traversed *unconditionally*; indeed, if there are multiple `Eps`-labelled transitions out of a state then any of them may lead to a successful match, so each, in principle, must be explored *non-deterministically*. The choice of which path to take is called 'non-deterministic' because we don't know in advance which one (if any) will lead to a successful match — in the worst case, we may have to try them all. The strings accepted by the NDA of Figure 1 are precisely those defined by the corresponding regular expression $(a \mid b)^*c$, e.g. `"ac"` (via the path

[2]Recall from Section 1 that null expressions are often denoted explicitly by the symbol ϵ, hence the name `Eps` used here.

$1 \to 5 \to 7 \to 8 \to 6 \to 3 \to 4 \to 2$), `"bc"` (path $1 \to 5 \to 9 \to 10 \to 6 \to 3 \to 4 \to 2$), `"aac"` (path $1 \to 5 \to 7 \to 8 \to 6 \to 5 \to 7 \to 8 \to 6 \to 3 \to 4 \to 2$) and so on.

Turning things around, we can ask whether a given input string will be accepted by an NDA by using the individual characters of that string to 'fire' the non-`Eps` transitions in the NDA, beginning from the start state. If all the characters of the input string have been used up and it is possible to reach the accepting state unconditionally from the current state then the input string is accepted by the NDA. If not, or if we reach a point where no transitions can fire, then the string is *rejected*. For example, the strings `"ac"`, `"bc"`, `"aac"` etc. will all be accepted. However, `"cc"` will be rejected, because we will reach the terminal state with the second `'c'` left unused. For the string `"ad"` the `'a'` will fire the transition from 7 to 8 but none of the successor transitions (states 4, 7 and 8) are labelled with a `'d'`, so this will also be rejected.

Note that a * in a regular expression creates a cycle in the corresponding NDA. For example, in Figure 1 the * in $(a \mid b)^*c$ is implemented via the "backward" transition from state 6 to 5. Note also that the "short-circuit" transition from 1 to 3 allows the NDA to accept `'a'` or `'b'` *zero* times, as required.

Remark: The NDAs shown here are not necessarily optimal, but are shown here as generated by the NDA construction algorithm that you will be implementing later on.

3.1 Representation

An automaton can be described using the following Haskell data types:

```
type State = Int

data Label = C Char | Eps
           deriving (Eq, Ord, Show)

type Transition = (State, State, Label)

type Automaton = (State, [State], [Transition])
```

States are labelled with unique integer identifiers. A transition comprises a source state, a target state and a label, which is either `Eps` (unconditional transition) or of the form `C c` where `c` is a character. An automaton is a 3-tuple comprising, in order:

1. The (unique) start state
2. The *list* of terminal states†
3. The list of transitions

† **Note**: Although an NDA has only one terminal state, in general an automaton can have many, as we shall see in Part IV.

As an example, the NDA of Figure 1 (`ndaFigure` in the template) will be represented by:

```
(1,[2],[(1,3,Eps),(1,5,Eps),(3,4,Eps),(4,2,C 'c'),(5,7,Eps),
        (5,9,Eps),(6,3,Eps),(6,5,Eps),(7,8,C 'a'),(8,6,Eps),
        (9,10,C 'b'),(10,6,Eps)])
```

and that of x'^* (`nda2` in the template) by:

```
(1,[2],[(1,3,C 'x'),(3,4,Eps),(4,2,Eps),(4,5,Eps),(5,6,C '\''),
        (6,2,Eps),(6,5,Eps)])
```

The order in which the states, terminals and transitions are listed is not significant, although we always display each of them sorted lexicographically. The NDAs for the examples given in Table 1 are shown graphically in Table 3. Note that the NDA for $(ab)?d^+$ corresponds to the simplified version of the expression in which the $^+$ and ? have been removed, *viz.* $(ab\,|\,)dd^*$.

4 What to do

There are four parts to this test and most of the marks are for Parts I–III. Part IV is worth only two of the 25 marks available and is hard, so you are advised to attempt it only when you have completed Parts I–III.

The example expressions referred to above, (`re1`, `re2`, ...) and their corresponding NDAs, (`nda1`, `nda2`, ...), are included in the template for testing purposes. There are also some examples of so-called *deterministic* automata (`da1`, `da2`, ...) each of which has the same structure as a non-deterministic automaton, but with possibly more than one terminal state. These feature in Part IV but are still useful elsewhere for testing purposes.

A function `showRE :: RE -> String` is also defined in the template for testing purposes; this produces a compact textual representation of a given `RE`. For example, `showRE re5`, where `re5` is `Seq (Opt (Seq (Term 'a') (Term 'b'))) (Plus (Term 'd'))`, produces `"(ab)?d+"`.

4.1 Part I: Basics

1. Define a function: `lookUp :: Eq a => a -> [(a, b)] -> b` that will look up a given item in a list of item/value pairs, delivering the corresponding value. A precondition is that there is exactly one occurrence of the item in the list.

   [1 mark]

2. Define a function `simplify :: RE -> RE` that will remove any $^+$ or ? expressions using the simplification rules in Section 2.1. For example, `showRE (simplify re5)` should produce `"(ab|)dd*"`.

   [3 marks]

4.2 Part II: Functions on NDAs

1. Define the following three functions for indexing automata:

   ```
   startState :: Automaton -> State
   terminalStates :: Automaton -> [State]
   transitions :: Automaton -> [Transition]
   ```

   that will return respectively the start state, the list of terminal states and the list of transitions in a given automaton. For example, `terminalStates nda1` should return `[2]`. Use these wherever you see fit in the questions that follow.

   [1 mark]

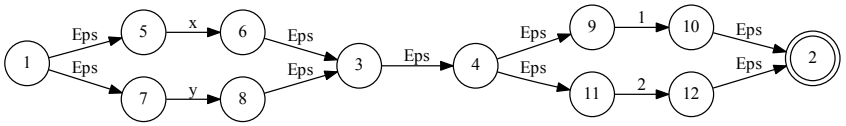
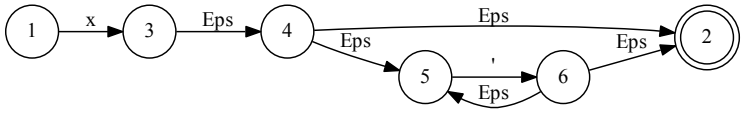
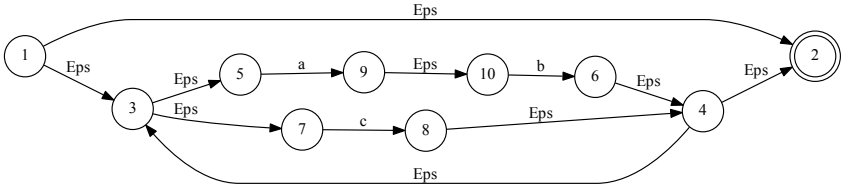
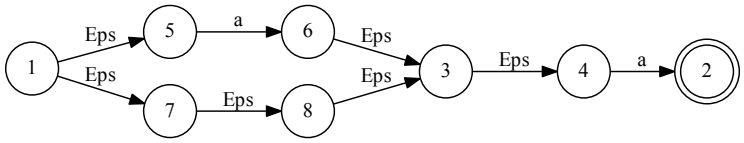
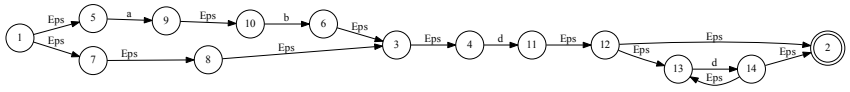
Regex	NDA
$(x \mid y)(1 \mid 2)$	
x'^*	
$(ab \mid c)^*$	
$(a \mid \epsilon)a$	
$(ab)?d^+$	

Table 3: NDAs for the regular expressions in Table 1

2. Define a function `isTerminal :: State -> Automaton -> Bool` that delivers true iff the given state is a terminal (accepting) state in the given automaton. For example, `isTerminal 2 nda1` and `isTerminal 3 da4` should both return `True` and `isTerminal 5 nda2` should return `False`.

[1 mark]

3. Define a function `transitionsFrom :: State -> Automaton -> [Transition]` that returns the list of transitions emanating from a given state in a given automaton. For example the application `transitionsFrom 5 ndaFigure` should return `[(5,7,Eps),(5,9,Eps)]` and `transitionsFrom 10 nda3` should return `[(10,6,C 'b')]`. If the state doesn't appear in the atomaton then the result should be `[]`.

[2 marks]

4. Define a function `labels :: [Transition] -> [Label]` that returns the labels in the given list of transitions, with no duplicates. Any `Eps` transitions should be excluded from the result. For example, `labels [(1,2,Eps)]` should return `[]` and `labels (transitions nda3)` should return `[C 'a',C 'c',C 'b']`. Hint: use `nub` from `Data.List` to remove any duplicates.

[1 mark]

5. Define a function `accepts :: Automaton -> String -> Bool` that returns `True` iff the given automaton accepts the given input string using the so-called "backtracking" approach, which is described below. Note that this is not the most efficient way to solve the problem, but it is the simplest.

[6 marks]

Acceptance testing using backtracking

The idea is to try to match all possible paths from a given state against the given input string. If *any* of them succeeds then the string is accepted. To implement this, define a helper function, `accepts' :: State -> String -> Bool`, that takes the current state, `s` say, (initially the unique start state) and the string we're trying to match (initially the input string), and does the following:

- If the state is a terminal state *and* the string is null then the string is accepted, i.e. there are no unmatched characters left. Use the `isTerminal` function defined earlier and the `null` function from the prelude.
- Otherwise, you need to use the `transitionsFrom` function to get the list of transitions emanating from `s` and then "try" each of them looking for a match; this is the "non-deterministic" bit. The match succeeds if *any* of these paths leads to the string being accepted; conversely, it fails if the string is rejected by every path followed. To implement this part of the solution it is suggested, although not required, that you define another helper function `try :: String -> Transition -> Bool` that 'tries out' one such transition using the following rules:
  - If the transition is labelled with an `Eps` with target state `t` then you simply continue the process from state `t` by invoking `accepts'` recursively; the string is unchanged.

– If the transition is labelled with a character, `c` say, with target state `t`, then you need to examine the input string (use pattern matching!). If the string is empty then the match fails at this point and we abort (return `False`). If it is non-empty then look at the item at the head, `c'` say. If `c' == c` then you continue by invoking `accepts'` on the tail of the string starting with state `t`; otherwise the match has again failed at this point and you return `False`.

4.3 Part III: Constructing an NDA

Define a function `makeNDA :: RE -> Automaton` that will generate an NDA from a given regular expression. There is a simple algorithm for doing this which you should try to follow exactly, in order to simplify testing and marking. We start with the top-level function, which is provided in the template thus:

```
makeNDA :: RE -> Automaton
makeNDA re
  = (1, [2], sort transitions)
  where
    (transitions, k) = make (simplify re) 1 2 3
```

Your job is to define the function `make`, which has type

```
make :: RE -> Int -> Int -> Int -> ([Transition], Int)
```

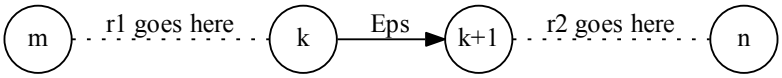
In addition to the expression we're converting, the function takes three integers, `m`, `n` and `k` say, in that order. The idea is to construct an NDA which starts in the state numbered `m` and ends in the state numbered `n`. The `k` represents the next available identifier that can be used to number any additional states. The top-level call defines the start state of the final NDA to be 1 and the end state to be 2, but this is just a convention – NDAs in general can have arbitrary start and terminal states. Because this is the top level, 2 must therefore be the unique terminating state for the whole NDA, hence the `[2]` in the resulting 3-tuple. Because state identifiers 1 and 2 have been used, the next available identifier is 3, hence the fourth parameter to `make`. The transitions returned by `make` are sorted lexicographically, using the `sort` function imported from the module `Data.List`, in order to help with testing and marking.

To `make` the NDA for a given regular expression you simply put together nodes and transitions according to the rules defined in Figure 2. The dotted lines indicate where the NDAs for any subexpressions should be placed; these sub-NDAs will be built recursively.

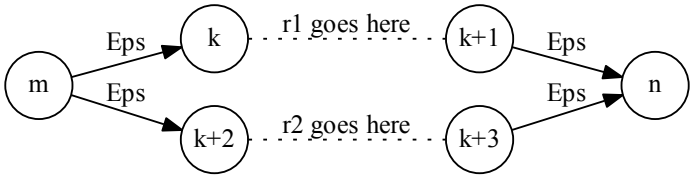
As an example, given an expression `Seq r1 r2` you recursively make the NDA for `r1` with start and end states `m` and `k` respectively, then do the same for `r2` with start and end states `k+1` and `n` respectively. You then add one additional transition labelled `Eps` between states `k` and `k+1` and you're done. You have to be careful to maintain the next available state identifier at each point: making the NDA for `r1` may consume an arbitrary number of such identifiers, for example. However, it tells you which ones it used as part of its return value, whose type is `(([Transition], Int)`. The `Int` here is the next available identifier, *having built the entire sub-NDA for* `re1`, as represented by the list of transitions. The idea is then to use this as the `k` parameter when making the NDA for `r2`. Once you see how this works for sequences, very similar rules apply for the other two recursive cases.

The base cases are simple. The result in each case is a *single* transition appropriately labelled and the next available state identifier is the one we started with, as no additional nodes are required.

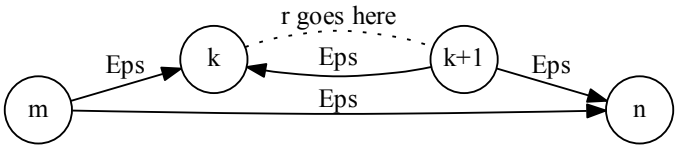
(a) Null and terminal cases: `Eps` and `C c`



(b) Sequence: `Seq r1 r2`



(c) Alternative: `Alt r1 r2`



(d) Repetition: `Rep r`

Figure 2: Rules for constructing an NDA

If you follow the state numbering scheme exactly as described then you should be able to reproduce the NDAs for each example expression in the template. For example `makeNDA re2` should yield, ideally exactly, if not the equivalent of, `nda2`, i.e.

```
(1,[2],[(1,3,C 'x'),(3,4,Eps),(4,2,Eps),(4,5,Eps),(5,6,C '\''),
        (6,2,Eps),(6,5,Eps)])
```

You can test the bases cases separately, e.g. `makeNDA (Term 'x')` should yield the 3-tuple `(1,[2],[(1,2,C 'x')])`, or the equivalent.

Note that an exact reproduction isn't necessary to get the marks, however; what matters is whether the NDA does the right thing.

[8 marks]

5 Part IV: Deterministic Finite Automata

The final part of this exercise invites you to remove the `Eps` transitions in the NDAs from Part III, generating a *Deterministic* Automaton, or DA, where the next transition, if one is possible, is uniquely determined by the transition labels. To illustrate what you need to achieve, Table 4 shows the deterministic equivalents of the NDAs in Table 3. These DAs are defined in the template and can be used for testing.

There are several ways of building a DA from an NDA. The algorithm below explains the general idea, together with details of a particular approach, *superset construction*, that falls out elegantly in Haskell that you may wish to follow. Feel free to try a different approach, however, if you think you have a better idea!

However you do it, the objective is to define a function `makeDA :: Automaton -> Automaton` that converts an NDA into a DA. Note that the representations of both NDAs and DAs are the same. The functions you defined earlier will thus work equally well on both.

Translating an NDA into a DA

The idea is for the states of the DA, the so-called "metastates", to correspond to *sets* of states in the original NDA. The first objective is to compute the list of metastates and transitions representing the DA, together with the root (start) metastate. Then all you need to do is map metastates into ordinary, integer-labelled, states and determine the *set* of terminal states – there may be more than one.

The suggestion is that you make the list of new metastates and new transitions accumulating parameters of a helper function. The algorithm then proceeds as follows:

- We work with *input sets* of states from the original NDA, represented as lists, beginning with the singleton set containing just the start state, e.g. `[1]` if the start state is 1. Now compute the *frontier* of the NDA, which is the list of non-`Eps` transitions that can be reached by following `Eps` transitions as far as possible, starting from each state in the input set. If the frontier reaches the terminal state then the special "phantom" transition `(t, t, Eps)` should be included in the list returned (see below). In order to avoid the need to do explicit cycle detection we'll assume a precondition that any cycle in the NDA must include at least one non-`Eps` transition – it will if the NDA has been built as described earlier. For example, the frontier of the NDA for $(ab\,|\,c)^*$, i.e. `nda3`, starting from input state set `[1]`, is `[(2,2,Eps),(5,9,C 'a'),(7,8,C 'c')]`. In general there may be several states in the input set, so simply concatenate the frontiers for each of them together. The recommendation is that you define a function `getFrontier :: State -> Automaton -> [Transition]` for calculating frontiers as described. The type signature for this function is provided in the template.

- The set of *unique* source states of each transition in the frontier defines a new "metastate", which will be associated with a state in the DA being constructed. For example, the metastate for the frontier above will be `[2,5,7]`. Note the role of the phantom transition: this

metastate will (later) be marked as a terminal state because it contains 2 - the unique terminal state in this particular NDA. Hint: use `nub` and `sort` imported from `Data.List` to ensure that the metastate "identifiers" are uniquely determined by the original state identifiers they contain.

- If the metastate has been visited before then return immediately as we've reached a state that we've seen before: the root is the metastate and the accumulated lists of metastates and transitions are unchanged.

- Otherwise, group the frontier transitions by transition label — there may be several transitions with the same label and these need to be combined (see below). If the frontier contains the special phantom transition then this should be removed at this point. In the example above the grouped transitions will be `[(C 'a',[9]),(C 'c',[8])]` as the labels are unique. However, the frontier of the NDA for $(a\,|\,)a$ from state 1 contains two `'a'` transitions, i.e. `[(3,7,C 'a'),(5,9,C 'a')]`, so these will be grouped thus: `[(C 'a',[7,9])]`. The recommendation is that you use the `labels` function defined earlier to define a function `groupTransitions :: [Transition] -> [(Label, [State])]` to group transitions as described, making sure that the states in the list (`[State]`) are unique. The type signature for this function is provided in the template. Hints: use a list comprehension and notice that the `labels` function will conveniently remove the phantom transition automatically, if it is present, because it is an `Eps` transition.

- You now add the new metastate, `m` say, to the accumulated list of metastates and recursively visit each target node set, adding one new transition to the accumulated transitions list after each recursive call. For example, if the grouped transitions are `[(C 'a',[5,6,7]),(C 'c',[8,9])]` you recurse with the state sets (lists) `[5,6,7]` and `[8,9]`, maintaining the list of new metastates and transitions as you go (a `fold`?!). If the first of these generates the root `r`, and transition and metastate lists `ts` and `ms` respectively, then you add one new transition of the form `(m, r, C 'a')` to `ts` before recursing again on the state list `[8,9]` and doing likewise.

If you choose to follow the algorithm as described you may like to use the following function types defined in the template:

```
type MetaState = [State]

type MetaTransition = (MetaState, MetaState, Label)

makeDA :: Automaton -> Automaton
   -- Suggested helper function for makeDA...
  makeDA' :: [State] -> [MetaState] -> [MetaTransition]
          -> (MetaState, [MetaState], [MetaTransition])
```

The second and third parameters of `makeDA'`, and similarly the result tuple, are the accumulating parameters suggested.

Good luck!

[2 marks]

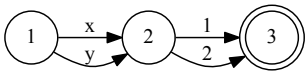
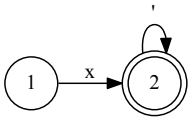
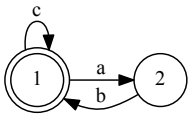
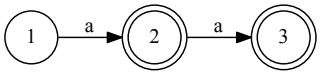
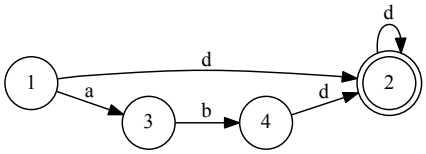
Regex	DA
$(x \mid y)(1 \mid 2)$	
x'^*	
$(ab \mid c)^*$	
$(a \mid \epsilon)a$	
$(ab)?d^+$	

Table 4: DAs for the examples in Table 1