Haskelleene: A Very Haskell Implementation of Regular Expression and Finite Automaton Equivalence

Liam Chung, Brendan Dufty, Lingyuan Ye

Wednesday 29$^{\text{th}}$ May, 2024

Abstract

In this project, we implement finite (non)deterministic automata and regular expressions, with their corresponding semantics of regular languages. We also implement the conversions between these different structures, including power-set determinisation and Kleene's algorithm. Lastly, we use QuickCheck to verify the behavioural equivalence of automata and regular expressions under these constructions, as stated by Kleene's Theorem.

Contents

0.1 The Alphabet

In this section we define our most basic data structure: a finite input alphabet. Our current implementation choice is to record alphabet as a *type class*, equipped with a complete list of symbols:

```
module Alphabet where

import Data.List ( sort )
import Test.QuickCheck ( elements, Arbitrary(arbitrary) )

class Ord a => Alphabet a where
  completeList :: [a]
```

The reason for this implementation choice is that we can silently pass this recorded list of complete alphabet as input via constraint declarations. We also require any alphabet shall be ordered.

Here is an example: The function `alphIter` will check if a list contains exactly each element of the alphabet once. This function will be useful when we work with deterministic automata.

```
alphIter :: Alphabet a => [a] -> Bool
alphIter l = sort l == completeList
```

The main input alphabet we are going to use on testing consists of three letters. This choice is of course not essential to our main implementation, which will be parametric over all type instances of the class `Alphabet`.

```
data Letter = A | B | C deriving (Eq, Ord)

instance Show Letter where
  show A = "a"
  show B = "b"
  show C = "c"

instance Alphabet Letter where
  completeList = [A,B,C]
```

To use the QuickCheck library to test on arbitrary inputs, we also define an instance of

0.2 Deterministic and Non-Deterministic Automata

Our first major goal is to implement (non)deterministic automata. Recall that an automaton with input alphabet Σ consists of a set of states, a subset of accepting states, and a transition function δ. As input, δ takes a symbol of the alphabet, and a current state of the automaton. For a deterministic automaton, it outputs a unique next state; For a non-deterministic automaton, it outputs a *list* of possible next states.

```
{-# LANGUAGE TupleSections #-}

module Automata where
```

```haskell
import Alphabet ( Alphabet(..), alphIter ) -- contains all of our utility functions
import Data.Maybe ( isJust , fromJust , isNothing )
import Data.List ( nub )
import qualified Data.Set as Set
```

However, since δ of an automaton is a *function*, it is not directly definable as inputs. Our implementation choice is to record the *transition table* of the transition function, and use the following types as an interface of encoding and decoding an automata with finite inputs:

```haskell
type TDict l s = [(s, [(Maybe l, s)])]

data AutData l s = AD { stateData :: [s]
                      , acceptData :: [s]
                      , transitionData :: TDict l s }

pPrintAD :: (Show l, Show s, Eq l, Eq s) => (AutData l s) -> String
pPrintAD ad = "States:" ++ showSts ad (stateData ad) ++ "\n\nTransitions:" ++ transitions (
    transitionData ad) where
    showSts _ [] = ""
    showSts d (s:ss) = " " ++ show s ++ isAccept s d ++ showSts d ss
    isAccept s d = if s `elem` acceptData d then "*" else ""
    transitions [] = ""
    transitions (t:ts) = "\n" ++ stTrs t ++ transitions ts
    stTrs (_, []) = ""
    stTrs (s, (t:ts)) = "\n" ++ show s ++ " --" ++ letter ++ "-> " ++ output ++ stTrs (s,ts)
        where
      letter | (fst t) == Nothing = "em"
             | otherwise = show $ fromJust (fst t)
      output = show $ snd t
```

Here `l` should be thought of as the type of the chosen alphabet, while `s` is the type of our states. The type `TDict` then acts as the type of transition tables. A pair in `TDict` should be thought of recoding the information of given a current state, what are the possible output states of a given input. The fact that we have used `Maybe l` is that we want the type `AutData` to be simultaneously able to record both deterministic and non-deterministic automata, thus with the possibility of ϵ-transitions. For some examples of using `AutData` to encode automata, see Section 2.1.

As mentioned, the data type of deterministic automata should be defined as follows:

```haskell
data DetAut  l s = DA { states :: [s]
                      , accept :: [s]
                      , delta :: l -> s -> s }
```

A consequence of using the same data type (`AutData`) to encode both deterministic and non-deterministic automata is that we need to check whether the given data properly defines an automaton of either type. The following are some useful utility functions to check whether the given transition table has certain properties:

```haskell
-- intended to be used as aut `trsOf` s, to see what transitions s has in aut
trsOf :: Eq s => AutData l s -> s -> [(Maybe l, s)]
trsOf aut s
  | isNothing $ lookup s $ transitionData aut = []
  | otherwise = fromJust $ lookup s $ transitionData aut

-- utility for checking if a list has duplicates
allUnq:: Eq a => [a] -> Bool
allUnq = unqHelp []
    where
```

```
        unqHelp _ [] = True
        unqHelp seen (x:xs) = notElem x seen && unqHelp (x:seen) xs

appendTuple :: ([a],[b]) -> ([a],[b]) -> ([a],[b])
appendTuple (l,l') (m,m') = (l++m,l'++m')
```

We will then use these functions to check if a set of automaton data can properly encode a deterministic automaton. To access a deterministic automaton from an element of `AutData`, we need to verify that the given transition table is indeed deterministic, i.e. for any current state, for any given input alphabet there exists a unique output state:

```
-- contingent on passing detCheck, turn data into DA
encodeDA :: (Alphabet l, Eq s) => AutData l s -> Maybe (DetAut l s)
encodeDA d | not $ detCheck d = Nothing
           | otherwise = Just $ DA { states = stateData d
                                   , accept = acceptData d
                                   , delta = safeDelta } where
    safeDelta ltr st = fromJust $ lookup (Just ltr) $ fromJust (lookup st (transitionData d))
    detCheck ad = length sts == length (stateData ad) && allUnq sts
                  -- all states are in transitionData exactly once
                  && all detCheckTr stateTrs where
                  -- check transitions for each letter exactly once
      sts = fst <$> transitionData ad
      stateTrs = snd <$> transitionData ad
    detCheckTr trs = notElem Nothing (fst <$> trs)
                     -- no empty transitions
                     && alphIter (fromJust . fst <$> trs)
                     -- transition set is exactly the alphabet
```

We end the basic implementation of deterministic automata by providing its semantic layer, i.e. on a given input and an initial state, whether a deterministic automata accept a list of symbols or not. Intuitively, the `run` function uses the transition function to trasverses an input string on an automaton and output the terminating state. Then `acceptDA` tests whether the input is accepted by testing whether the terminating state is an accepted state:

```
-- takes DA, input letter list, and initial state to output pair
run :: DetAut l s -> s -> [l] -> s
run _ s0 [] = s0
run da s0 (w:ws) = run da (delta da w s0) ws

acceptDA :: (Eq s) => DetAut l s -> s -> [l] -> Bool
acceptDA da s0 w = run da s0 w `elem` accept da
```

We now proceed to implement non-deterministic automata, in a very similar way. The only difference is that the transition function now also accepts empty input, viz. the socalled ϵ-*transitions*, and the result of a transition function is a list of all possible next states.

```
data NDetAut l s = NA { nstates :: [s]
                      , naccept :: [s]
                      , ndelta :: Maybe l -> s -> [s] }
```

Completely similarly, we can encode a non-determinisitic automaton from an element of `AutData`: Maybe say more.

```
-- make data into an NA (if data formatted properly, go ahead; otherwise, format it then encode
    )
encodeNA :: (Alphabet l, Eq s) => AutData l s -> NDetAut l s
```

```haskell
encodeNA d = NA { nstates = stateData d
                , naccept = acceptData d
                , ndelta = newDelta } where
  newDelta sym st = case lookup st tData of
                      Nothing -> []
                      Just ls -> nub [ st' | (sym', st') <- ls, sym' == sym, isJust sym' || st'
                           /= st ]
  tData = if trsMerged rawTData then rawTData else mergeTrs rawTData
  trsMerged = allUnq . map fst
  rawTData = transitionData d
  -- slow, so we don't always want to be calling this
  mergeTrs [] = []
  mergeTrs ((tr0,tr1):trs) = mTr:mergeTrs remTrs where
    mTr = (tr0, fst prop ++ tr1)
    remTrs = snd prop
    prop = propTrs tr0 trs
  -- for a given state, propagate all of its outputs together, and return
  propTrs _ [] = ([],[])
  propTrs st (tr:trs) = appendTuple resultTuple (propTrs st trs) where
    resultTuple = if st == fst tr then (snd tr,[]) else ([],[tr])

-- put an NA back into autdata, e.g. to turn it into regex
decode :: (Alphabet l, Eq s) => NDetAut l s -> AutData l s
decode nda = AD { stateData = sts
                  , acceptData = naccept nda
                  , transitionData = trandata
                  }
  where sts = nstates nda
        ntrans = ndelta nda
        symlist = Nothing : (Just <$> completeList)
        trandata = graph help sts
        help st = concatMap (\sym -> (sym,) <$> ntrans sym st) symlist
        graph f as = zip as $ f <$> as
```

We end with the semantic layer for non-deterministic automata. The algorithm used for implementing `runNA` for trasversing an input string on a non-deterministic automaton is inspired by [Cox07]. Intuitively, we record a list of *active states* at each step of the trasversal, with its corresponding remaining list of inputs. If there are no possible transition states with the given input, we terminiate and record it in the output. The function `ndautAccept` then checks whether there is an output that consumes all the inputs, and terminiates at an accepting state.

```haskell
runNA :: (Alphabet l, Ord s) => NDetAut l s  -> s -> [l] -> [([l], s)]
runNA na st input =
  case input of
    [] -> ([],) <$> epReachable (ndelta na) st
    (w:ws) -> concatMap (\s -> runNA na s input) nsucc ++
              case wsucc of
                [] -> [(input,st)]
                ls -> concatMap (\s -> runNA na s ws) ls
      where wsucc = ndelta na (Just w) st
    where   nsucc = ndelta na Nothing  st

ndautAccept :: (Alphabet l, Ord s) => NDetAut l s -> s -> [l] -> Bool
ndautAccept na s0 w = any (\(ls,st) -> null ls && st `elem` naccept na) $
                      runNA na s0 w

ndfinalStates :: (Alphabet l, Ord s) => NDetAut l s -> s -> [l] -> [s]
ndfinalStates na s0 w = snd <$> runNA na s0 w
```

Finally, we implement an algorithm that exhibits the behavioural equivalence between deterministic and non-deterministic automata. The easy direction is that, evidently, any deterministic automaton is also a non-deterministic one, and they evidently accepts the same language:

```haskell
-- The trivial forgetful functor: DA -> NA
fromDA :: (Alphabet l) => DetAut l s -> NDetAut l s
fromDA da = NA { nstates = states da
               , naccept = accept da
               , ndelta = newDelta } where
  newDelta Nothing _ = []
  newDelta (Just l) st = [delta da l st]
```

As an aside, we can now actually create `Show` instances for both `DetAut` and `NDetAut`, but decoding them to `AutData` and using the `Show` instance we defined for that:

```haskell
pPrintNA :: (Alphabet l, Show l, Show s, Eq l, Eq s) => NDetAut l s -> String
pPrintNA = pPrintAD . decode

pPrintDA :: (Alphabet l, Show l, Show s, Eq l, Eq s) => DetAut l s -> String
pPrintDA = pPrintAD . decode . fromDA
```

Now returning to automaton conversion: the non-trivial direction is that any non-deterministic automaton can also be converted into a deterministic one, with possibly different set of states and transition functions. The general idea is simple: We change the set of states to the set of *subset* of the original non-determinisitic automaton. This way, we may code the non-deterministic behaviour in a deterministic way. The algorithm is inspired by [Fio10].

```haskell
-- The Power-set Construction: NA -> DA
fromNA :: (Alphabet l, Ord s) => NDetAut l s -> DetAut l (Set.Set s)
fromNA nda = DA { states = Set.toList dasts
                , accept = Set.toList $ Set.filter acchelp dasts
                , delta = fromTransNA ntrans
                }
  where ndasts = nstates nda
        dasts  = Set.powerSet $ Set.fromList ndasts
        ndaacc = naccept nda
        acchelp set = not $ Set.disjoint set $ Set.fromList ndaacc
        ntrans = ndelta nda

epReachable :: (Alphabet l, Ord s) => (Maybe l -> s -> [s]) -> s -> [s]
epReachable ntrans st = st : concatMap (epReachable ntrans)
                                       (ntrans Nothing st)

fromTransNA :: (Alphabet l, Ord s) => (Maybe l -> s -> [s]) ->
                                      l -> Set.Set s -> Set.Set s
fromTransNA ntrans sym set = result
  where starts = listUnions (epReachable ntrans) set
        step = listUnions (ntrans $ Just sym) starts
        result = listUnions (epReachable ntrans) step
        listUnions f input = Set.unions $ Set.map Set.fromList $ Set.map f input

fromStartNA :: (Alphabet l, Ord s) => NDetAut l s -> s -> Set.Set s
fromStartNA nda st = Set.fromList $ epReachable ntrans st
  where ntrans = ndelta nda
```

We have tested the behavioural equivalence using the above transitions in Section 2.

0.3 Regular Expression Library

```haskell
module Regex where
```

In this section, we will define regular expressions, in the Kleene algebraic sense of the term. It's important to note that this version of regular expressions is different from those that are well known to programmers. For example, the language $\{a^n b a^n | n \in \mathbb{N}\}$ is well known to not be regular, and so not have a regular expression that represents it; meanwhile the programmer's regular expressions can encode this language rather easily.

The following serves as our definition of the `Regex` type. First we define our base case constructors, `Empty`, `Epsilon`, and `L l`. Note the distinction between `Empty` and `Epsilon` type constructors. The former is the regex representing the empty language, that is, the language that has no words in it. The latter represents the empty string, which is the word with no letters, and as a regular expression is the string language containing one string: the empty string.

Note also that we use a type parameter `l` for this type. This is so that we can use different input alphabets if we so choose; see the `Alphabet` module for the definition of the `Alphabet` type class in Section 0.1.

```
data Regex l = Empty |
               Epsilon |
               L l |
               Alt (Regex l) (Regex l) |
               Seq (Regex l) (Regex l) |
               Star (Regex l)
  deriving (Eq, Show)
```

We also write a robust pretty printing function for `Regex`, with many hard coded cases so as to mimic the conventions of writing regular expressions as we can. We then also include some quality-of-life functions, for example for sequencing or alternating a list of regexes, as well as doing so for some list of input letters. This allows us to turn a word into a regex representing exactly that word quickly and easily.

```
pPrintRgx :: Show l => Regex l -> String
pPrintRgx Empty = "em"
pPrintRgx Epsilon = "ep"
pPrintRgx (L a) = show a
pPrintRgx (Alt (Seq r r') (Seq r'' r''')) = "(" ++ pPrintRgx (Seq r r') ++ ")+("
                                             ++ pPrintRgx (Seq r'' r''') ++ ")"
pPrintRgx (Alt (Seq r r') r'') = "(" ++ pPrintRgx (Seq r r') ++ ")" ++ "+" ++ pPrintRgx r''
pPrintRgx (Alt r'' (Seq r r')) = pPrintRgx r'' ++ "+" ++ "(" ++ pPrintRgx (Seq r r') ++ ")"
pPrintRgx (Alt r r') = pPrintRgx r ++ "+" ++ pPrintRgx r'
pPrintRgx (Seq (Alt r r') (Alt r'' r''')) = "(" ++ pPrintRgx (Alt r r') ++ ")("
                                             ++ pPrintRgx (Alt r'' r''') ++ ")"
pPrintRgx (Seq (Alt r r') r'') = "(" ++ pPrintRgx (Alt r r') ++ ")" ++ pPrintRgx r''
pPrintRgx (Seq r'' (Alt r r')) = pPrintRgx r'' ++ "(" ++ pPrintRgx (Alt r r') ++ ")"
pPrintRgx (Seq r r') = pPrintRgx r ++ pPrintRgx r'
pPrintRgx (Star (L a)) = "(" ++ show a ++ "*)"
pPrintRgx (Star r) = "(" ++ pPrintRgx r ++ ")*"

-- QoL functions for sequencing or alternating lists of regexes
seqList :: [Regex l] -> Regex l
seqList [l] = l
seqList (l:ls) = Seq l $ seqList ls
seqList [] = Epsilon

altList :: [Regex l] -> Regex l
altList [l] = l
altList (l:ls) = Alt l $ altList ls
altList [] = Empty
```

```
-- QoL functions for turning lists of letters into sums or products
seqList', altList' :: [l] -> Regex l
seqList' = seqList . map L
altList' = altList . map L
```

The proof system for the equational theory of regular expressions, known as Kleene algebra, can reason about equivalence of regular expressions, for example:

$$a + (b + a^*) = a + (a^\star + b) = (a + a^\star) + b = a^\star + b$$

It is outside of the scope of this project to implement a proof searcher for this system. Having said that, there are certain simplifications we can make to regular expressions that will help improve readability and running time. For example:

$$\emptyset + r = r + \emptyset = r$$
$$\epsilon r = r\epsilon = r$$

...and more. The objective here is not to simplify the regular expression as far as possible, but to implement easy simplifications that improve readability (limiting occurrence of redundancies, and so on).

```
simplifyRegex :: Eq l => Regex l -> Regex l
simplifyRegex regex | helper regex == regex = regex
                    | otherwise = helper regex where
 helper rx =
  case rx of
    Alt r4 (Seq r1 (Seq (Star r2) r3))
      | r1 == r2 && r3 == r4 -> Seq (Star (simplifyRegex r1)) (simplifyRegex r4)
    (Alt Empty r) -> simplifyRegex r
    (Alt r Empty) -> simplifyRegex r
    (Alt r r') | r == r' -> simplifyRegex r
    (Seq r Epsilon) -> simplifyRegex r
    (Seq Epsilon r) -> simplifyRegex r
    (Seq _ Empty) -> Empty
    (Seq Empty _) -> Empty
    (Star Empty) -> Epsilon
    (Star Epsilon) -> Epsilon
    (Star (Star r)) -> simplifyRegex $ Star r
    (Star (Alt r Epsilon)) -> simplifyRegex $ Star r
    (Star (Alt Epsilon r)) -> simplifyRegex $ Star r
    Alt r r' -> Alt (simplifyRegex r) (simplifyRegex r')
    Seq r r' -> Seq (simplifyRegex r) (simplifyRegex r')
    Star r -> Star (simplifyRegex r)
    x -> x
```

Now we need to set to the task of defining a semantics for these regular expressions. That is, given a list of letters from the input alphabet (a word), check whether it belongs to the language represented by the regular expression. First, we will need a utility function for checking if initial sequences of the word satisfy part of the regex, specifically for the `Sequence` and `Star` cases.

This function takes a `Regex` and a word, and produces all splits of the word where the first part of the split satisfies the regex. By splits of a word, we mean splitting the word into two subwords, that when concatenated give the original word. For example splits of *abc* are:

$$[(abc, \epsilon),\ (ab, c),\ (a, bc),\ (\epsilon, abc)]$$

and for this particular input word, with the regex $c^\star a^\star$, `initCheck` would output (ϵ, abc) and (a, bc). Note that this function does use `regexAccept`, which we will define below, and that this function does nothing to reduce the "size" of r, which means we need to be careful about infinite looping. More on that below.

```
initCheck :: Eq l => Regex l -> [l] -> [([l],[l])]
initCheck r w = filter (regexAccept r . fst) $ splits w where
  splits [] = [([],[])]
  splits (x:xs) = map (appFst x) (splits xs) ++ [([],x:xs)]
  appFst x (y,z) = (x:y,z)
```

Now we finally define the semantics for our regular expressions. Our base cases are simple: the empty language accepts no words, the ϵ accepts only the empty word, and l for some letter accepts only that letter. We also include special cases for when sequencing with a single letter: because our sequencing checker will be pretty inefficient, and this is a common use case that is quite fast, we encode it directly. The `Alt` case is also straightforward, effectively just acting as a disjunction in the most simple of terms.

As mentioned above, our sequencing and star operations are slower in general; this is because we need to examine all initial sequences of the word to see if they can match the regex. This is why `initCheck` returns the split of the word, rather than just the initial segment that matches: we need to then check the rest of the regex (the other operand in the case of `Seq`, or the regex again in the case of `Star`) on what remains of the string.

POTENTIAL OPTIMISATION: encode maximal length of a regex and compare to length of the input string, to find easier reject cases. also, don't split up front: do it as we go, this will make longer words accept faster.

```
regexAccept :: Eq l => Regex l -> [l] -> Bool
-- the empty language accepts no words
regexAccept Empty _    = False
-- if down to the empty string, only accept the empty word
regexAccept Epsilon [] = True
regexAccept Epsilon _  = False
-- if down to a single letter, only accept that letter (and if longer, reject too)
regexAccept (L _) []   = False
regexAccept (L l) [c]  = l == c
regexAccept (L _) _ = False
-- optimisations for simple sequences (one part is just a letter)
regexAccept (Seq (L _) _) [] = False
regexAccept (Seq _ (L _)) [] = False
regexAccept (Seq (L l) r) (c:cs) = l == c && regexAccept r cs
regexAccept (Seq r (L l)) cs = last cs == l && regexAccept r (init cs)
regexAccept (Seq Epsilon r) cs = regexAccept r cs
-- general Alt case pretty easy
regexAccept (Alt r r') cs = regexAccept r cs || regexAccept r' cs
-- general Seq case is less efficient
regexAccept (Seq r r') cs = any (regexAccept r' . snd) $ initCheck r cs
-- if word is empty, star is true
regexAccept (Star _) [] = True
-- general star case
regexAccept (Star r) cs =
  any (regexAccept (Star r) . snd) $ ignoreEmpty $ initCheck r cs
  where ignoreEmpty = if regexAccept r [] then init else id
```

In the general case of `Star`, similar to `Seq`, we want to find all initial segments of the word that satisfy the regular expression; but now we try to proceed using `Star r` again. There is an important, subtlety, however: we want to avoid infinite looping, which may happen if our regular expression accepts the empty word.

Take for example the regex $(\epsilon + a)^\star$. This is equivalent to $a^\star$, of course, but introducing these kinds of simplifications to `simplifyRegex` significantly increases the complexity. If we're not careful, inputting the string bbb with this regex will loop infinitely because our we will continually find that (ϵ, bbb) satisfies the regex, and will loop back and forth between `regexAccept` and `initCheck` without making any forward progress in matching the word. To avoid this, we check if the inner regex accepts ϵ. If it does, we know that it is redundant (because of our case where `regexAccept (Star r) [] = True`) and so we use `init` to drop the last accepting initial segment: given our implementation of `initCheck`, this will necessarily be (ϵ, w) for whatever input word w.

1 Automata and Regular Expressions

We have now defined (non)deterministic automata and regular expressions. Next, perhaps unsurprisingly, since these are well known to be two sides of the same coin, we encode a method to translate between them. **Possible to do:** and prove these operations are inverses of each other. Converting from a regex to a non-deterministic automaton is relatively straightforward, so we will begin with that. Second, we will describe Kleene's algorithm (a variation of the Floyd-Warshall Algorithm) in order to transform an automaton into a regex. Finally, we will note general problems with the above two steps as well as possible future methods to improve them.

```
module Kleene where

import Alphabet   ( Alphabet(..) )
import Automata   ( AutData(..), DetAut(..), NDetAut(..), encodeNA, trsOf )
import Regex      ( Regex(..), simplifyRegex, altList, seqList )
import Data.Maybe ( fromJust, isNothing )

import qualified Data.Map.Strict as Map

type RgxAutData l = (AutData l Int, Int)
```

1.1 Regular Expressions to Automata

Converting a regular expression into a non-deterministic automaton is both straightforward and (mostly) intuitive. Because regular expressions are defined inductively, we can associate an automaton with each base case, and then associate methods for combining automata for inductive cases. At a high level we can think of our algorithm as follows: first, the transition labels correspond to our alphabet; second, generate a very simple automaton for each atom (letter, epsilon, or empty string); then attach these automata together in a well-behaved way for each operator. Roughly, `Seq` corresponds to placing each automaton one after the other, `Alt` to placing them in parallel, and `Star` to folding the automaton into a circle. We will explain each of these operations more clearly in the appropriate section. As for any inductive construction we start with our base cases. Note these choices are not unique (there are many automata that accept no words) but we go with the simplest ones for each case:

`Empty` : a single, non-accepting, state.

`Epsilon` : a single, accepting, state.

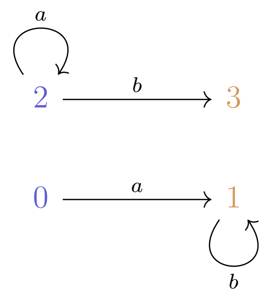
`L a` : a non-accepting state, connected to an accepting one by a

```
regToAut :: Regex l -> RgxAutData l
-- gives an integer automata and a starting state
regToAut Empty = (AD [1] [] [], 1)
regToAut Epsilon = (AD [1] [1] [], 1)
regToAut (L l) = (AD [1,2] [2] [(1, [(Just l,2)])], 1)
regToAut (Seq a b) = seqRegAut (regToAut a) (regToAut b)
regToAut (Alt a b) = altRegAut (regToAut a) (regToAut b)
regToAut (Star a)  = starRegAut $ regToAut a

-- convenience function for encoding result directly to an NA
fromReg :: (Alphabet l) => Regex l -> (NDetAut l Int, Int)
fromReg reg = (encodeNA ndata,st)
  where (ndata,st) = regToAut reg
```

As we head into each inductive step, we note a few conventions. First, an observant reader will have seen that our function outputs automata data with integer states. Since we are inductively constructing automata we need to be adding new states while preserving the old ones (and their transition functions). `Int` states make it very easy to relabel them, and—as we will see later—make it much easier to run algorithms on. Below is the function for the `Seq` operator, alongside a helpful fetch function.

```
seqRegAut :: RgxAutData l -> RgxAutData l -> RgxAutData l
seqRegAut (aut1,s1) (aut2,s2) =
  (adata, 13*s1) where
  adata = AD
    ([ x*13 | x <- stateData aut1 ] ++ [ x*3 | x <- stateData aut2 ]) -- states
     [ 3*x | x <- acceptData aut2 ]                                   -- accepting states
     (gluingSeq (aut1, s1) (aut2, s2))                                -- transition function

gluingSeq :: RgxAutData l -> RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingSeq (aut1, _) (aut2, s2) =  fstAut ++ mid ++ sndAut where
  fstAut = [mulAut 13 x (aut1 `trsOf` x) | x <- stateData aut1, x `notElem` acceptData aut1]
  mid = [mulAut 13 x ((aut1 `trsOf` x)++[(Nothing,3*s2)])  | x <- acceptData aut1]
  sndAut = [mulAut 3 x (aut2 `trsOf` x) | x <- stateData aut2]
  mulAut n x trs = (x*n, multTuple n trs)
```

This function takes two automata `aut1,aut2` and glues them together by adding epsilon transitions between the accepting states of `aut1` and the starting state of `aut2`. We need to add these epsilon transitions rather than merely identify the starting/ending in order to preserve transitions out of said states. For example, if we identify states 1 and 2 in the following two automata (blue states being intial, ornage being accepting):



we would change the language from ab^*a^*b to $a(b^*a^*)^*b$. Thus, we add epsilon transitions.

Additionally, we multiply the states in the first automaton by 13 and states in the second automaton by 3. In the gluing and star operator we have to add new states (in order to prevent the gluing issue above). We always add a state labeled 1 for a starting state and 2 for an accepting state. Multiplication by prime numbers allows us to ensure that our new automaton *both* preserves the transition function of its component parts *and* has distinct state labels for every state. Each input for each operator has a unique prime number assigned to it:

1. `Seq`: $13, 3$
2. `Alt`: $5, 7$
3. `Star`: 11.

```
altRegAut :: RgxAutData l -> RgxAutData l -> RgxAutData l
altRegAut (aut1, s1) (aut2, s2) =
  (adata, 1) where
  adata =  AD
    ([1,2] ++ [ x*5 | x<- stateData aut1 ] ++ [ x*7 | x<- stateData aut2 ])
    [2]
    (gluingAlt (aut1, s1) (aut2, s2))

gluingAlt :: RgxAutData l -> RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingAlt (aut1,s1) (aut2,s2) = start ++ fstAut ++endFstAut ++ sndAut ++ endSndAut where
  start = [(1, [(Nothing, s1*5), (Nothing, s2*7)])]
  fstAut = [mulAut 5 x (aut1 `trsOf` x) | x <- nonAccept aut1]
  sndAut = [mulAut 7 x (aut2 `trsOf` x) | x <- nonAccept aut2]
  endFstAut = [appSnd (Nothing,2) (mulAut 5 x (aut1 `trsOf` x)) | x <- acceptData aut1]
  endSndAut = [appSnd (Nothing,2) (mulAut 7 x (aut2 `trsOf` x)) | x <- acceptData aut2]
  mulAut n x trs = (x*n, multTuple n trs)
  nonAccept aut = [ x | x <- stateData aut, x `notElem` acceptData aut ]
  appSnd a (b,c) = (b,a:c)
```

Our construction for `Alt` is defined similarly to `Seq`. We add a new initial and acceptance state to ensure that our gluing preserves the appropriate transition functions. Lastly, we define our `Star` construction as follows (alongside some helper functions):

```
starRegAut :: RgxAutData l -> RgxAutData l
starRegAut (aut, s) =
  (adata, 1) where
  adata = AD
    (1:[x*11 | x<- stateData aut])
```

```
    [1]
    (gluingStar (aut, s))

gluingStar :: RgxAutData l -> [(Int, [(Maybe l, Int)])]
gluingStar (aut1, s1) = start ++ middle ++ end where
  start = [(1, [(Nothing, s1*11)])]
  middle = [(x*11, multTuple 11 (aut1 `trsOf` x))| x<-stateData aut1, x `notElem` acceptData
      aut1]
  end = [(a*11, (Nothing, 1) : multTuple 11 (aut1 `trsOf` a)) | a <- acceptData aut1]

multTuple :: Int -> [(a,Int)] -> [(a,Int)]
multTuple _ [] = []
multTuple n ((a,b):xs) = (a,n*b) : multTuple n xs

addTuple :: Int -> [(a,Int)] -> [(a,Int)]
addTuple _ [] = []
addTuple n ((a,b):xs) = (a,n+b) : multTuple n xs
```

For `Star` we add a single state which serves as both the initial state and an accept state. By connecting the beginning and ending of our starting automaton we create an abstract loop—corresponding to the operation `Star`.

This construction was relatively straightforward since by looking at what each operator in a regular expression means an automaton immediately suggests itself. The next algorithm, moving from automata to regular expressions, is far less intuitive, and encounters difficulties we will note in the final section. This complexity is due to the non-inductive/recursive definition of automata as opposed to regex.

1.2 Automata to Regular Expressions

Here, we implement which take a non/deterministic automaton, a starting state, and outputs a corresponding regular expression. The algorithm we use, called Kleene's algorithm, allows us to impose a semi-recursive structure on an automaton which in turn allows us to extract a regular expression. First, we provide the implement of Kleene's algorithm (as well as some motivation and examples) before explaining how Kleene's algorithm can provide us with our desired conversion. We conclude with a brief overview of the helper functions we enlist throughout our implementation as well as a slightly different conversion (and why we opted with our method.)

1.2.1 Kleene's Algorithm

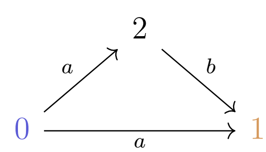
Below, you will find our implementation of Kleene's algorithm; it take an automaton (whose states are labeled $[0..n]$ exactly), and three integer i, j, k (which correspond to states) and outputs a regular expression corresponding to the set of all paths from state i to state j without passing through states higher than k. This is a rather strong structural requirement, but it allows us to define the algorithm recursively and—as we will show later in the report—it is easy to convert any automaton into one with the correct state labels.

```
kleeneAlgo :: Eq l => AutData l Int -> Int -> Int -> Int -> Regex l
```

```haskell
kleeneAlgo aut i j (-1) =
  if i==j
  then simplifyRegex (Alt Epsilon (altList [ maybe Epsilon L a | a <- successorSet aut i j]))
  else simplifyRegex (altList [ maybe Epsilon L a | a <- successorSet aut i j])
kleeneAlgo aut i j k =
  simplifyRegex (Alt (simplifyRegex $ kleeneAlgo aut i j (k-1))
                     (simplifyRegex $ seqList
                              [ simplifyRegex $ kleeneAlgo aut i k (k-1)
                              , simplifyRegex $ Star $ kleeneAlgo aut k k (k-1)
                              , simplifyRegex $ kleeneAlgo aut k j (k-1) ]))
```

Let us quickly dig in what this code actually means before moving onto an example. The algorithm successively removes states by decrementing k. At each step we remove the highest state and nicely add its associate transition labels to the remaining states. If `regToAut` worked by building up an automaton to follow a regular expression, Kleene's algorithm works by pulling a fully glued automaton apart step by step. When $k = -1$, we want to return a regex corresponding to the set of paths from i directly to j without stopping at any either state along the way. This is simply the set of transition labels which connect i to j (alongside `Epsilon` if $i = j$.) However, if $k > -1$, we need to remove the k'th state and shift the transition functions into and out of k amidst the rest of the automaton. First, we don't touch any of the paths which avoid k by including `kleeneAlgo aut` i j $(k-1)$. The remaining sequence can be viewed as: take any path to you want to k but stop *as soon as* you reach k for the first time; then, take any path from k to k as many times as you want (we need the Star here because this algorithm does not normally permit loops); finally, take any path from k to j. As we will see in the following example, this entire process can be though of as a single transition label encoding all of the data that used to be at k. By removing every state, we are left with a single arrow which corresponds to our desired regular expression.

In the following toy example, we apply the algorithim to this automata with intial state 0 and accepting state 1:



We apply the algorithim to 0 (the starting state) 1 (the accepting state) and 2 (the largest state.) By removing 2 from the state space (which is what incrementing k does), we get

$$(a\epsilon^* b) + a$$

which is exactly what the regular expression corresponding to this automaton is. Hopefully this very simple example has highlighted the motivation and process behind the algorithim.

With this broad motivation, we can know discuss how to implement the algorithm to provide our desired conversion:

```haskell
autToReg :: Eq l => Ord s => (AutData l s, s)-> Regex l
autToReg (aut, s)= altList [kleeneAlgo intAut firstState a lastState | a <- acceptData intAut]
    where
  intAut = fst $ relabelAut (aut,s)
  firstState = relabelHelp aut s
  lastState = length (stateData intAut) - 1
```

This takes an automaton (and a starting state), transforms that automaton into one with the appropriate state labels and then applies the algorithm on the initial state and every accepting state. For a given accepting state a, (`kleeneAlgo aut` firstState a lastState) provides a regular expression corresponding to the paths from the initial state to a with no restrictions - we have set k to be higher than every state label. This is exactly what we were looking for, given our previous understanding of the algorithm itself.

Below we briefly describe the helper functions needed for this implementation as well as an alternate definition of autToReg, whose problems we will expound upon in the last section of this chapter.

```
-- takes automata data and states s1, s2. Outputs all the ways to get s2 from s1
successorSet :: Eq s => AutData l s -> s -> s -> [Maybe l]
successorSet aut s1 s2
  | isNothing (lookup s1 (transitionData aut) ) = [] -- if there are no successors
  | otherwise = map fst (filter (\w -> s2 == snd w) (fromJust $ lookup s1 (transitionData aut))
      )
```

This function simply returns returns all the ways to directly move between two states for the base case of Kleene's algorithm.

```
-- states as integer makes everything easier. We use a dictionary to relabel in one sweep per
    say
relabelHelp :: Ord s => AutData l s -> s -> Int
relabelHelp aut s = fromJust (Map.lookup s (Map.fromList $ zip (stateData aut) [0 .. (length (
    stateData aut))]))

relabelAut :: Ord s => (AutData l s, s) -> RgxAutData l
relabelAut (aut, s1) = (AD [relabelHelp aut s | s <- stateData aut]
                           [relabelHelp aut s| s<- acceptData aut]
                           [(relabelHelp aut s, [ (a, relabelHelp aut b)| (a,b) <- aut `trsOf`
                               s] ) | s<- stateData aut]
                         , relabelHelp aut s1)
```

As mentioned, we need to convert an arbitrary automaton to one with well-behaved state labels in order to define Kleene's algorithm. These functions do so handily via the use of a dictionary.

```
-- take aut data and make a nice start state/end state
cleanAutomata:: RgxAutData l -> RgxAutData l
cleanAutomata (aut, s) = (AD (0:lastState:[x+1 | x <- stateData aut])
                             [lastState]
                             (cleanTransition (aut,s))
                          ,0) where lastState = 1+length (stateData aut)

cleanTransition:: RgxAutData l -> [(Int, [(Maybe l, Int)])]
cleanTransition (aut, s) = start ++ middle ++ end where
  start = [(0, [(Nothing, s+1)])]
  middle = [(x+1, addTuple 1 (aut `trsOf` x)) | x <- stateData aut, x `notElem` acceptData aut]
  end = [(x+1, (Nothing, length (stateData aut) + 1) : addTuple 1 (aut `trsOf` x)) | x<-
      acceptData aut]
```

AUTTOREGSLOW WAS REMOVED HERE, DIDN'T REMOVE ANY OTHER TEXT These last pieces of code allow us to define a version of autToReg which takes in multiple initial states rather than just one. It does so by adding a new initial (and accepting) state after the relabeling - connected via epsilon transition. While this construction is more general, it adds several more transitions which further increase the size of the corresponding regular expression. More on this issue in the following section.

```haskell
-- Another implementation of Automata to Reg, assuming the aut is deterministic
dautToReg :: (Alphabet l, Ord s) => DetAut l s -> s -> Regex l
dautToReg daut s = simplifyRegex $ foldr (Alt . dautToRegSub daut s (states daut)) Empty $
    accept daut where
  dautToRegSub da s0 [] sn = if s0 /= sn then resut else Alt Epsilon resut
    where trans = delta da
          succs = filter (\l -> trans l s0 == sn) completeList
          resut = foldr (Alt . L) Empty succs
  dautToRegSub da s0 (s1:ss) sn = simplifyRegex $ Alt reg1 $ Seq reg2 $ Seq (Star reg3) reg4
    where reg1 = simplifyRegex $ dautToRegSub da s0 ss sn
          reg2 = simplifyRegex $ dautToRegSub da s0 ss s1
          reg3 = simplifyRegex $ dautToRegSub da s1 ss s1
          reg4 = simplifyRegex $ dautToRegSub da s1 ss sn
```

1.3 Issues with the Algorithms

The most prominent issue with this algoritihm is that it creates very complex regular expressions—each step adds a `Seq`, `Alt`, and `Star`. We have attempted to implement a few simplification throughout the algorithm, but it still outputs expressions that are horribly over-complex. For example,

$$\begin{aligned}&\texttt{autToReg}\ (\texttt{wikiAutData}, 0) = \\ &\quad b + c + ((\epsilon + a)(a^*)(b + c)) + \\ &\qquad ((b + c + ((\epsilon + a)(a^*)(b + c))) \\ &\qquad\ (\epsilon + b + ((a + c)(a^*)(b + c)))^* \\ &\qquad\ (\epsilon + b + ((a + c)(a^*)(b + c))))\end{aligned}$$

which, upon manual reduction, is equivalent to

$$(a + c)^* b(b + c + a(a + c)^* b)^*$$

However, reduction of regular expressions is NP-hard, and so we have simply tried to encode a few, computationally quick, simplifications as noted in Section 0.3 This is most pressing when we convert back into an automaton, since each new operator corresponds to an additional structural level in the automaton and an additional computational complication we need to overcome when running words on automata. There are several ways this could be improved, but they fall beyond the scope of this report. Perhaps most straightforward we could further improve our regular expression simplifcation—or change the inductive construction to more easily allow for commutativity. Furthermore, their are additional algorithms beyond just Kleene's for converting from automata to regular expressions, searching for definining properties to make an automaton a better fit for another may be a helpful optimisation. Finally, we could instead simplify the autmata rather than the regular expression. There exists a minimal equivalent determinisitic automata for every regular language, implementing such a minimisation algorithm would also increase usability.

2 Testing

2.1 Examples

In this section we define several examples of (non)deterministic automata and regular expressions, and run tests on them to verify the correctness of our algorithms written earlier. More concretely, we would like to test:

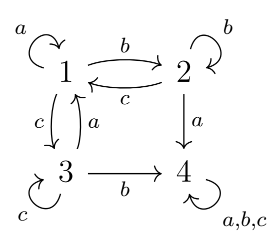
- All the basic semantic layers for (non)deterministic automata and regular expressions should be correctly working.
- The conversion between deterministic and non-deterministic automata implemented in Section 0.2 should be behaviourally equivalent.
- The conversion between (non)deterministic automata and regular expressions implemented in Section 1 should be behaviourally equivalent.

2.1.1 Examples of Automata and Regular Expressions

We begin by including the relevant modules.

```
module Examples where

import Alphabet ( Letter(..) )
import Automata
    ( NDetAut,
      DetAut,
      AutData(AD),
      encodeDA,
      run,
      acceptDA,
      encodeNA,
      decode,
      runNA,
      ndautAccept )
import Regex ( Regex(..) )
import Kleene ( autToReg, dautToReg )
import Data.Maybe ( fromJust )
```

Now we give some examples of automata. Our first is the following deterministic automaton:

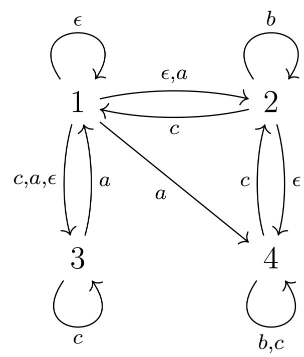


```haskell
myAutData :: AutData Letter Int
myAutData = AD [1,2,3,4]         -- the states
               [4]               -- accepting states
               [(1,[(Just A,1)   -- the transitions
                   ,(Just B,2)
                   ,(Just C,3)])
               ,(2,[(Just A,4)
                   ,(Just B,2)
                   ,(Just C,1)])
               ,(3,[(Just A,1)
                   ,(Just B,4)
                   ,(Just C,3)])
               ,(4,[(Just A,4)
                   ,(Just B,4)
                   ,(Just C,4)])]

myDA :: DetAut Letter Int
myDA = fromJust $ encodeDA myAutData

myDAtoReg :: Regex Letter
myDAtoReg = dautToReg myDA 1

 -- automaton taken from Wikipedia Page on Kleenes Algorihtm
wikiAutData :: AutData Letter Int
wikiAutData = AD [0,1]
                 [1]
                 [(0, [(Just A, 0)
                      ,(Just B, 1)
                      ,(Just C, 0)])
                 ,(1, [(Just A, 0)
                      ,(Just B, 1)
                      ,(Just C, 1)])]

wikiDA :: DetAut Letter Int
wikiDA = fromJust $ encodeDA wikiAutData

wikiDAtoReg :: Regex Letter
wikiDAtoReg = dautToReg wikiDA 0
```

By manually checking, the following should be an accepting input for `myDA`.

```haskell
-- an accepting sequence of inputs
myInputs :: [Letter]
myInputs = [A,A,A,A,B,C,B,B,B,A]

myTestRun :: (Int, Bool)
myTestRun = (finalst, result)
  where finalst = run myDA 1 myInputs
        result = acceptDA myDA 1 myInputs
```

Let us also consider this example of a non-deterministic automaton:




```
myNAutData :: AutData Letter Int
myNAutData = AD [1,2,3,4]          -- the states
                [4]                -- accepting states
                [(1,[(Nothing,2)
                    ,(Just C,3)])
                ,(2,[(Nothing,4)
                    ,(Just B,2)
                    ,(Just C,1)])
                ,(1,[(Just A,2), -- want to merge these with above
                     (Just A,3)])
                ,(3,[(Just A,1)
                    ,(Just C,3)])
                ,(1,[(Nothing,1)  -- want to be ignoring this
                    ,(Nothing,3)  -- want to merge these with above
                    ,(Just A,4)])
                ,(4,[(Just B,4)
                    ,(Just C,4)])]

-- this automaton, encoded
myNDA :: NDetAut Letter Int
myNDA = encodeNA myNAutData
```

Here are is an example for regular expressions:

```
exampleRegex :: Regex Letter
exampleRegex = Star (Alt (L A) (L B))
```

2.2 QuickCheck

We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where

import Test.Hspec ( hspec, describe, it, shouldBe )
import Test.Hspec.QuickCheck ( prop )
import Test.QuickCheck ( (==>) )
import Automata ( acceptDA, decode, fromNA, fromStartNA, ndautAccept )
import Regex ( regexAccept )
import Kleene ( autToReg, dautToReg, fromReg )
import Examples ( exampleRegex, myNDA, myTestRun, wikiDA )
```

We have tested the following basic facts:

- A basic running example for a deterministic automaton.
- The behavioural equivalence of determinisitic and non-deterministic automata under the conversion implemented in Section 0.2.
- The behavioural equivalence of regular expressions and its corresponding non-deterministic automaton implemented in Section 1.
- The behavioural equivalence of a deterministic automaton and its corresponding regular expression implemented in Section 1.
- The behavioural equivalence of a non-deterministic automaton and its corresponding regular expression implemented in Section 1.

```
main :: IO ()
main = hspec $ do
  describe "Examples" $ do
    it "DA test run result should be (4,True)" $
      myTestRun `shouldBe` (4,True)
    prop "NA and transfer to DA should give the same result" $
      \input -> ndautAccept myNDA 1 input == acceptDA (fromNA myNDA) (fromStartNA myNDA 1)
          input
    prop "reg to NA should give the same result" $
      \input -> regexAccept exampleRegex input == uncurry ndautAccept (fromReg exampleRegex)
          input
    prop "DA to reg should give the same result" $
      \input -> length input <= 30 ==>
                acceptDA wikiDA 0 input == regexAccept (dautToReg wikiDA 0) input
    prop "NA to reg should give the same result" $
      \input -> length input <= 30 ==>
                ndautAccept myNDA 1 input == regexAccept (autToReg (decode myNDA, 1)) input
```

The result is recorded below. The reason in the last two cases we restrict the arbitrarily generated input data to have length less than 30 is that the current algorithms is not efficient enough to run the semantics for larger inputs on regular expressions.

```
Examples
  DA test run result should be (4,True) [\/]
  NA and transfer to DA should give the same result [\/]
    +++ OK, passed 100 tests.
  reg to NA should give the same result [\/]
    +++ OK, passed 100 tests.
  DA to reg should give the same result [\/]
    +++ OK, passed 100 tests; 84 discarded.
  NA to reg should give the same result [\/]
    +++ OK, passed 100 tests; 84 discarded.

Finished in 6.8299 seconds
5 examples, 0 failures
```

3 Conclusion

References

[Cox07] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). `https://swtch.com/~rsc/regexp/regexp1.html`, 2007.

[Fio10] Marcelo Fiore. *Lecture Notes on Regular Languages and Finite Automata*. Accessible at `https://www.cl.cam.ac.uk/teaching/1011/RLFA/LectureNotes.pdf`, 2010.