Haskelleene: An Implementation of Regular Expression and Automata Equivalence

Liam Chung, Brendan Dufty, Lingyuan Ye

Sunday 26$^{\text{th}}$ May, 2024

Abstract

In this document, we implement (non)deterministic automata and regular expressions, with their corresponding semantics on input strings. We also implement the well-known conversion between automata and regular expressions, and use QuickCheck to verify their behavioural equivalence.

Contents

1 Basic Library

1.1 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 Basics 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 the type `Letter` as an `Arbitrary` instance:

```
instance Arbitrary Letter where
  arbitrary = elements completeList
```

1.2 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

import Basics ( Alphabet(..), alphIter ) -- contains all of our utility functions
import Data.Maybe ( isJust, fromJust )
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:

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

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

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 3.

A consequence of using the same data type to possibly encode both a deterministic and non-deterministic automaton is that, we need further conditions to check whether the given data is well-structured to give an output of what we want. The following are some useful utility functions to check whether the given transition table has certain properties:

```
-- given data in the format of transitionData, an input state, and input
-- letter, output all possible input/output pairs.
getTrs :: (Eq a, Eq b) => [(a,[(b,a)])] -> a -> b -> [(b,a)]
getTrs allTrs s0 ltr = filter (\x -> fst x == ltr) $ filter (\x -> fst x == s0) allTrs >>= snd

-- 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')
```

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

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

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:

```
-- check the totality and functionality of the transition table
detCheck :: (Alphabet l, Eq s) => AutData l s -> Bool
detCheck ad = length sts == length (stateData ad) && allUnq sts
              -- all states are in transitionData exactly once
              && all detCheckHelper stateTrs where
              -- check transitions for each letter exactly once
  sts = fst <$> transitionData ad
  stateTrs = snd <$> transitionData ad

detCheckHelper :: (Alphabet l, Eq s) => [(Maybe l,s)] -> Bool
detCheckHelper trs = notElem Nothing (fst <$> trs)
                     -- no empty transitions
                     && alphIter (fromJust . fst <$> trs)
                     -- transition set is exactly the alphabet

-- contingent on passing safetyCheck, make data into a 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))
```

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
```

Completely similarly, we implement non-deterministic automata. The only difference now 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
encodeNA d = NA { nstates = stateData d
                , naccept = acceptData d
                , ndelta = newDelta } where
```

```haskell
  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 :: Eq s => TDict l s -> TDict l s
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 output together, and return
-- all of them, as well as the transition data with those removed
-- type TDict l s = [(s, [(Maybe l, s)])]
propTrs :: Eq s => s -> TDict l s -> ([(Maybe l,s)], TDict l s)
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-determinisitic 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:

```
-- 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]
```

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].

```
-- 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 transition in Section 3.

1.3 Regular Expression Library

This is where we define the Kleene regular expression structure we will be using throughout the project.

```
module Regex where

import Data.Maybe ( isNothing )
-- -----------------
-- REGEX definitions
```

```
-- -----------------

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

-- showing a regular expression. Lots of hard-coded cases to make specific nicer and more
    readable
instance Show l => Show (Regex l) where
  show Empty = "âĹĚ"
  show Epsilon = "ÉŻ"
  show (L a) = show a
  show (Alt (Seq r r') (Seq r'' r''')) = "(" ++ show (Seq r r') ++ ")+(" ++ show (Seq r'' r''')
       ++ ")"
  show (Alt (Seq r r') r'') = "(" ++ show (Seq r r') ++ ")" ++ "+" ++ show r''
  show (Alt r'' (Seq r r')) = show r'' ++ "+" ++ "(" ++ show (Seq r r') ++ ")"
  show (Alt r r') = show r ++ "+" ++ show r'
  show (Seq (Alt r r') (Alt r'' r''')) = "(" ++ show (Alt r r') ++ ")(" ++ show (Alt r'' r''')
      ++ ")"
  show (Seq (Alt r r') r'') = "(" ++ show (Alt r r') ++ ")" ++ show r''
  show (Seq r'' (Alt r r')) = show r'' ++ "(" ++ show (Alt r r') ++ ")"
  show (Seq r r') = show r ++ show r'
  show (Star (L a)) = "(" ++ show a ++ "*)"
  show (Star r) = "(" ++ show r ++ ")*"
-- if sticking with altl, seql, then this isn't quite right. need paren cases

-- splits a list into all possible (order preserving) divisions of it
-- e.g. [1,2,3] becomes [([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]
splits :: [a] -> [([a],[a])]
splits [] = [([],[])]
splits (x:xs) = map (appendFst x) (splits xs) ++ [([],x:xs)]


-- append to the front of the first of a pair of lists
appendFst :: a -> ([a],[b]) -> ([a],[b])
appendFst x (y,z) = (x:y,z)

-- 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

-- very simple regex simplifications (guaranteed to terminate or your money back)
simplifyRegex :: Eq l => Regex l -> Regex l
simplifyRegex 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) -> Empty
                     (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

-- (Star (Alt (Alt Epsilon (L a)) (Seq (Alt (L b) (L c)) (Seq (Star (L b)) (Alt (L a) (L c)))))
    ,1)

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 Seq case is less efficient
regexAccept (Seq r r') cs = any (regexAccept r' . snd) $ trueInitCheck r cs
-- general Alt case pretty easy
regexAccept (Alt r r') cs = regexAccept r cs || regexAccept r' cs
-- if word is empty, star is true
regexAccept (Star _) [] = True
-- general star case
regexAccept (Star r) cs | regexAccept r [] = any (regexAccept (Star r) . snd) $ initCheck r cs
                        | otherwise = any (regexAccept (Star r) . snd) $ trueInitCheck r cs

-- get all initial sequences of the word that match the regex, except for
-- the empty initial sequence (as this loops forever and goes nowhere)
initCheck :: Eq l => Regex l -> [l] -> [([l],[l])]
initCheck r w = filter (regexAccept r . fst) $ init $ splits w

trueInitCheck :: Eq l => Regex l -> [l] -> [([l],[l])]
trueInitCheck r w = filter (regexAccept r . fst) $ splits w

-- these may eventually be useful so i won't delete them, but they're
-- not getting used right now. --LC

-- what is the maximum length of a string satisfying this regex?
-- Nothing means infinite (in other words, we used a star)
maxLenStr :: Regex l -> Maybe Int
maxLenStr Empty = Just 0
maxLenStr Epsilon = Just 0
maxLenStr (L _) = Just 1
maxLenStr (Alt r r') = maxMaybe (maxLenStr r) (maxLenStr r')
  where maxMaybe x y | isNothing x = x
                     | isNothing y = y
                     | otherwise = max x y
maxLenStr (Seq r r') = sum <$> sequence [maxLenStr r, maxLenStr r']
maxLenStr (Star _) = Nothing

loopSimpRgx :: Eq l => Regex l -> (Regex l,Int)
loopSimpRgx r | simplifyRegex r == r = (r,0)
              | otherwise = (next, count)
  where next = fst loop
```

```
        count = 1 + snd loop
        loop = loopSimpRgx (simplifyRegex r)
```

2 Automata and Regular Expressions

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

2.1 Regular Expressions to Automata

Converting a regular expression into a non-dterminsitic automata is both straightforward and (mostly) intuitive. We have defined a regular expression using an inductive constructor, similar to propositional logic. This inductive construction readily allows us to recursive construct this an automata - with one general construction per regex operator. At a high level we can think of our algorithim as follows: first, the transition labels correspond to our alphabet; second, generate a very simple automata for each atom (letter, epsilon, or empty string); then attach these automata together in a well-behaved way for each operator. Roughly, sequence corresponds to placing each automata one after the other, alternate to placing them in parralel, and star to folding the automata into a circle. We will explain each of these operations more clearly at the appropriate section. As for any inductive construction we need our base cases, which, for a regular expression are: the Empty expression, the Epislon expression, and a single letter. The simplest NDA which accepts no words is a single node with no accept states, a single node with a single accepting state accepts the empty word, and two nodes connected by a transition function labeled with our letter accepts said single letter.

```
module Kleene where

import Basics       -- contains all of our utility functions
import Automata     -- contains automaton definitions/conversions
import Regex        -- contains regex definitions
import Data.Maybe

import qualified Data.Map.Strict as Map

-- ---------------
-- REGEX to DetAut
-- ---------------

-- Since regex to automata inducts on the transition functions, we need a way to glue or
    reshape our automata nicely
-- As we add more states we need to relable them. The base states are 1 and 2, multiplying by
    3, 13 is sequence, multiplying by 5 and 7 is Alt, multiplying by 11 is Star
```

```
--useful function to glue states together


-- For each operator we define two corresponding functions. One outputs the automata data
    associated with that operator,
-- the other stitches and modifies the transition function across the inputs automata (mostly
    using Epsilon transitions)

regToAut :: Regex l -> (AutData l Int, Int)
-- 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

fromReg :: (Alphabet l) => Regex l -> (NDetAut l Int, Int)
fromReg reg = (encodeNA ndata,st)
  where (ndata,st) = regToAut reg
```

As we had 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). Integer states make it very easy to relable them, and - as we will see later - make it much easier to run algorithims on. Below, you can see our function for the Seq operator alongside a helpful fetch function.

```
seqRegAut :: (AutData l Int, Int) ->
             (AutData l Int, Int) ->
             (AutData l Int, Int)
seqRegAut (aut1,s1) (aut2,s2) =
  ( 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
  , 13*s1) -- starting state

gluingSeq :: (AutData l Int, Int) -> (AutData l Int, Int) ->
             [(Int, [(Maybe l, Int)])]
gluingSeq (aut1, _) (aut2, s2) =  firstAut ++ middle ++ secondAut where
  firstAut = [(13*x, multTuple 13 (transForState aut1 x)) | x<- stateData aut1, x `notElem`
      acceptData aut1]
  middle = [(13*x, multTuple 13 (transForState aut1 x) ++ [(Nothing, 3*s2)]) | x<- acceptData
      aut1]
  secondAut = [(x*3, multTuple 3 (transForState aut2 x)) | x<- stateData aut2]

transForState :: Eq s => AutData l s -> s -> [(Maybe l, s)]
transForState aut s
  | isNothing $ lookup s $ transitionData aut = []
  | otherwise = fromJust $ lookup s $ transitionData aut
```

This functoion 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 tranitions out of said states. For example, if we glued the start/accepting states together in the following automata DIAGRAM we would accept abaa, where FINISH EXAMPLE. Additionally, we multiply the states in the first automata by 13 and states in the second automata 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* had distinct state labels for every state. Each input for each operator has a unique prime number assigned to it:

1. Sequence: $13, 3$
2. Alternate: $5, 7$
3. Star: 11.

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

gluingAlt :: (AutData l Int, Int)->(AutData l Int, Int) ->
             [(Int, [(Maybe l, Int)])]
gluingAlt (aut1,s1) (aut2,s2) = start ++ firstAut ++endFirstAut ++ secondAut ++ endSecondAut
    where
  start = [(1, [(Nothing, s1*5), (Nothing, s2*7)])]
  firstAut = [(x*5, multTuple 5 (transForState aut1 x)) | x <- stateData aut1, x `notElem`
      acceptData aut1]
  secondAut = [(x*7, multTuple 7 (transForState aut2 x)) | x <- stateData aut2, x `notElem`
      acceptData aut2]
  endFirstAut = [(x*5, (Nothing,2) : multTuple 5 (transForState aut1 x)) | x <- acceptData aut1
      ]
  endSecondAut = [(x*7, (Nothing,2) : multTuple 7 (transForState aut2 x)) | x <- acceptData
      aut2]
```

Our construction for alternate is defined similarly to Sequence. 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 :: (AutData l Int, Int) -> (AutData l Int,Int)
starRegAut (aut, s) =
  ( AD
    (1:[x*11 | x<- stateData aut])
    [1]
    (gluingStar (aut, s))
  , 1)

gluingStar :: (AutData l Int, Int)-> [(Int, [(Maybe l, Int)])]
gluingStar (aut1, s1) = start ++ middle ++ end where
  start = [(1, [(Nothing, s1*11)])]
  middle = [(x*11, multTuple 11 (transForState aut1 x))| x<-stateData aut1, x `notElem`
      acceptData aut1]
  end = [(a*11, (Nothing, 1) : multTuple 11 (transForState aut1 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 node which serves as both the intitial state and an accept state. By connecting the beginning and ending of our starting automaton we create an abstract loop - which clearly corresponds to Star. Now that we have defined each construction, we provide a brief proof of the following lemma:

Lemma 2.1. *Each regular expression r gives rise to (at least one) non-deterministic automata, D, such that*

$$L(r) = L(D).$$

Proof. Let r be an arbitrary regular expression and set $aut = (fst.fromReg)r$. First, consider some $w \in L(r)$ and proceed via induction to show that $w \in L(aut)$.

We believe the base cases are clear from construction and so move one

□

This construction was relatively straightforward since by looking at what each operator in a regular expression means an automata immediately suggests itself. The next algorithim, 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/recurisve definition of automata as opposed to regex.

2.2 Automata to Regular

Here, we implement which take a non/determinisitc automata, a starting state, and ouputs a corresponding regular expression. The algorithim we use, called Kleene's Algorithim, allows us to impose a semi-recusrive structure on an automata which in turn allows us to extract a regular expression. First, we provide the implement of Kleene's Algorithim (as well as some motivation and examples) before explaining how Kleene's Algorithim 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.)

2.2.1 Kleene's Algorithim

Below, you will find our implementation of Kleene's Algorithim; it take an automata (whose states are labelled $[0..n]$ 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 algorithim recurivsely and - as we will show later in the report - it is easy to convert any automata into one with the correct state labels.

```
kleeneAlgo:: Eq l => AutData l Int -> Int -> Int -> Int -> Regex l
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 algoritihim succsessively removes states by incrementing k downwards. At each step we remove the highest state and nicely add its associate transition labels to the remaining states. If our regex to automata algoritihm worked by building up an automata to follow a regular expression, Kleene's Algorithm works by pulling a fully glued automata 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 automata. First, we don't touch any of the paths which avoid k by including $kleeneAlgoautij(k-1)$. 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 algorithim 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 lable 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.

TIKZ EXAMPLE.

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

```
-- Take a collection of data and starting states, outputs a regular expression which
    corresponds to the language. This version creates a nice first and last state
autToRegSlow :: Eq l => Ord s => (AutData l s, s)-> Regex l
autToRegSlow (aut, s)= kleeneAlgo intAut 0 lastState lastState where
  intAut = (fst.cleanAutomata.relabelAut) (aut,s)
  lastState = length (stateData intAut) - 1
--autToReg aut [s]= kleeneAlgo newAut 0 (length stateData aut) (length stateData aut) where
    newAut = cleanAutomata . relabelAut aut

--This version does not make a nice first and last state (and thus can't be adapted to multiple
     start states), but cuts down on the epislons
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

--following the Wikipedia page, this function recursively removes elements and uses the removed
     transition lables to construct the regex.
```

This takes an automata (and a starting state), transforms that automata into one with the appropriate state labels and then applies the algorithim on the intial 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 intial 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 algoritihim 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 some automata data and two states, s1, s2. Ouputs 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 Kleenes Algorithim.

```
-- 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) -> (AutData l Int, Int)
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) <-
                               transForState aut s] ) | s<- stateData aut]
                       , relabelHelp aut s1)
```

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

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

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

These last peices of code allow us to define a version of autToReg which takes in multiple intitial states rather than just one. It does so by adding a new intitial (and accepting) state after the relabelling - 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
-- We assume the automata 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

dautToRegSub :: (Alphabet l, Ord s) => DetAut l s -> s -> [s] -> s -> Regex l
dautToRegSub daut s0 [] sn = if s0 /= sn then resut else Alt Epsilon resut
  where trans = delta daut
        succs = filter (\l -> trans l s0 == sn) completeList
        resut = foldr (Alt . L) Empty succs
dautToRegSub daut s0 (s1:ss) sn = simplifyRegex $ Alt reg1 $ Seq reg2 $ Seq (Star reg3) reg4
  where reg1 = simplifyRegex $ dautToRegSub daut s0 ss sn
        reg2 = simplifyRegex $ dautToRegSub daut s0 ss s1
        reg3 = simplifyRegex $ dautToRegSub daut s1 ss s1
        reg4 = simplifyRegex $ dautToRegSub daut s1 ss sn
```

2.3 Issues with the Algorithims

Notes on why it took big and why it cant be fixed

3 Examples and Testing

```haskell
module Examples where

import Basics
import Automata
import Regex
import Kleene
import Data.Maybe

-- ----------------------
-- DETERMINISTIC AUTOMATA
-- ----------------------

-- Automata

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)])]

myDACheck :: Bool
myDACheck = detCheck myAutData
```

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

myDAtoReg :: Regex Letter
myDAtoReg = dautToReg myDA 1

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

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

wikiDAtoReg :: Regex Letter
wikiDAtoReg = dautToReg wikiDA 0

-- 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

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)])]

myNDA :: NDetAut Letter Int
myNDA = encodeNA myNAutData

myNDAtoReg :: Regex Letter
myNDAtoReg = autToReg (decode myNDA, 1)

-- what epsilon transitions does 1 have? should just be [2,3]
nothingFrom1 :: [Int]
nothingFrom1 = ndelta myNDA Nothing 1

myNInputsFalse :: [Letter]
myNInputsFalse = [B,B,A]

myNInputsTrue :: [Letter]
myNInputsTrue = []

myNTestRunFalse :: ([([Letter],Int)],Bool)
myNTestRunFalse = (filist,result)
  where filist = runNA myNDA 1 myNInputsFalse
```

```
        result = ndautAccept myNDA 1 myNInputsFalse

myNTestRunTrue :: ([([Letter],Int)],Bool)
myNTestRunTrue = (filist,result)
  where filist = runNA myNDA 1 myNInputsTrue
        result = ndautAccept myNDA 1 myNInputsTrue

exampleAut :: AutData String Int
exampleAut = AD [1,2,3] [2] [(1, [(Just "a", 1), (Just "b", 2)]), (2, [(Just "b", 2), (Just "a"
    , 3)  ]), (3, [(Just "a", 2), (Just "b", 2)])]

exampleAut2 :: AutData String Int
exampleAut2 = AD [1,2,3,4] [4] [(1, [(Just "a", 2)]), (2, [(Just "a", 3)]), (3, [(Just "a", 4)
    ])]

-- Regexes

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

annoyingRegex :: Regex Letter
annoyingRegex = Alt Empty (Seq Epsilon (L A))

-- examples
abc,abca,aOrbc :: Regex Letter
abc = seqList' [A,B,C]
abca = seqList' [A,B,C,A]
aOrbc = Seq abc $ Star (Alt (L A) (Seq (L B) (L C)))
```

4 Wrapping it up in an exectuable

We will now use the library form Section 1 in a program.

```
module Main where

import Basics

main :: IO ()
main = do
  putStrLn "Hello!"
```

We can run this program with the commands:

```
stack build
stack exec myprogram
```

5 Simple Tests

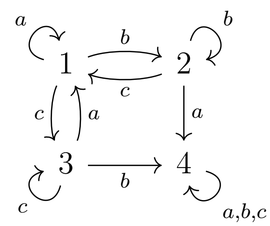
We now use the library QuickCheck to randomly generate input for our functions and test some properties.

```
module Main where
```

```
-- import Data.Maybe
import Test.Hspec
import Test.Hspec.QuickCheck
import Test.QuickCheck
-- import Basics
import Automata
import Regex
import Kleene
import Examples
```

The following uses the HSpec library to define different tests. Note that the first test is a specific test with fixed inputs. The second and third test use QuickCheck.

Here is the example deterministic automaton that is below:




```
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
```

6 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.