

CS454 Project 1: « Lexer Analysis »

M. Barney, J. Conrad, and S. Patel

March 14, 2013

1 Introduction

This is the final report for project 1, CS454, on lexical analysis.

Our first design decision was to use Haskell's literate mode to prepare all of our code. Secondly, we decided to use the distributed revision control software `git` for collaborative coding.

We decided to write each algorithm in the assignment as its own module, in addition to modules describing finite state automata (FSA) and regular expressions.

2 Finite State Automaton

In this module we give our data structure for modelling a finite state automaton.

The formal definition of an FSA is a 5-tuple, where:

1. a finite set of states (Q)
2. a finite set of input symbols called the alphabet (Σ)
3. a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
4. a start state ($q_0 \in Q$)
5. a set of accept states ($F \subset Q$)

We tried to have our data structure mirror the mathematical definition of an FSA as closely as possible.

```

{-# LANGUAGE TypeFamilies, FlexibleContexts, FlexibleInstances #-}
module FiniteStateAutomata (
    FSA (..),
    NFA' (..),
    NFAMap,
    DEA' (..),
    DFAMap,
    epsilon, ppfsa) where

import qualified Data.Map as M
import qualified Data.Set as S
type DFAMap a = M.Map Int (M.Map a Int)
type NFAMap a = M.Map Int (S.Set (Maybe a, Int))
class (Show (Elem m))  $\Rightarrow$  Listable m where
    type Elem m
    toList :: m  $\rightarrow$  [(Elem m, Int)]
instance (Show a)  $\Rightarrow$  Listable (M.Map a Int) where
    type Elem (M.Map a Int) = a
    toList = M.toList
instance (Show a)  $\Rightarrow$  Listable (S.Set (Maybe a, Int)) where
    type Elem (S.Set (Maybe a, Int)) = Maybe a
    toList = S.toList
class (Ord (Alpha f),
    Show (Alpha f),
    Show f,
    Show (FSAVal f),
    Listable (FSAVal f))  $\Rightarrow$  FSA f where
    type Alpha f
    type FSAVal f
    alphabet :: (Ord (Alpha f), Show (Alpha f))  $\Rightarrow$ 
        f  $\rightarrow$  S.Set (Alpha f)
    accepting :: f  $\rightarrow$  S.Set Int
    start     :: f  $\rightarrow$  Int
    trans     :: f  $\rightarrow$  M.Map Int (FSAVal f)
    states    :: f  $\rightarrow$  S.Set Int
    states fsa = S.unions [(S.fromList  $\circ$  M.keys  $\circ$  trans $ fsa),
        (accepting fsa),

```

```

    (S.fromList ◦
     concatMap sndList ◦
     M.elems ◦ trans $ fsa)]
sndList :: Listable m ⇒ m → [Int]
sndList = map snd ◦ toList
fsaShow :: (FSA f) ⇒ f → String
fsaShow fsa = "{alphabet="
    ++ (show ◦ S.toList ◦ alphabet $ fsa)
    ++ ", " ++
    "states=" ++
    (show ◦ S.toList ◦ states $ fsa) ++ ", " ++
    "start=" ++ (show ◦ start $ fsa) ++ ", " ++
    "accepting="
    ++ (show ◦ S.toList ◦ accepting $ fsa)
    ++ ", " ++ "trans="
    ++ (show ◦ map (filter (≠ ' ')) ◦
        showTransitions $ fsa)
pettyPrinter :: (FSA f) ⇒ f → IO ()
pettyPrinter fsa = (putStr $ "alphabet="
    ++ (show ◦ S.toList ◦ alphabet $ fsa)
    ++ "\n" ++
    "states="
    ++ (show ◦ S.toList ◦ states $ fsa)
    ++ "\n" ++
    "start=" ++ (show ◦ start $ fsa)
    ++ "\n" ++
    "accepting="
    ++ (show ◦ S.toList ◦ accepting $ fsa)
    ++ "\n") >> trans
    where trans =
        mapM_ (putStrLn ◦ filter (≠ ' '))
            $ showTransitions fsa
ppfsa :: (FSA f) ⇒ f → IO ()
ppfsa = pettyPrinter
showTransitions :: (FSA f) ⇒ f → [String]
showTransitions fsa = map showTransition ◦
    M.toList ◦ trans $ fsa where

```

```

showTransition (from, ts) = (show from)
  ++ " :: "
  ++ (show ∘ map showTransition' ∘ toList $ ts) where
    showTransition' (x, to) = (show x) ++ " -> " ++ (show to)

data DFA' a = DFA' { alpha :: S.Set a,
  ss :: DFAMap a,
  accept :: S.Set Int,
  st :: Int }

instance (Ord a, Show a) ⇒ FSA (DFA' a) where
  type Alpha (DFA' a) = a
  type FSAVal (DFA' a) = (M.Map a Int)
  alphabet = alpha
  accepting = accept
  start = st
  trans = ss

instance (Ord a, Show a) ⇒ Show (DFA' a) where
  show dfa = "DFA " ++ (fsaShow dfa)

data NFA' a = NFA' { nalpha :: S.Set a,
  nss :: NFAMap a,
  naccept :: S.Set Int,
  nst :: Int }

epsilon :: Maybe a
epsilon = Nothing

instance (Ord a, Show a) ⇒ FSA (NFA' a) where
  type Alpha (NFA' a) = a
  type FSAVal (NFA' a) = (S.Set (Maybe a, Int))
  alphabet = nalpha
  accepting = naccept
  start = nst
  trans = nss

instance (Ord a, Show a) ⇒ Show (NFA' a) where
  show nfa = "NFA " ++ (fsaShow nfa)

simpleNFA :: NFA' Char
simpleNFA = NFA' alpha states accepting start where
  alpha = S.fromList ['a', 'b']
  states = M.fromList

```

```

      [(0, S.fromList [(Just 'a', 1)]),
       (1, S.fromList [(Just 'b', 0), (epsilon, 2)])]
start = 0
accepting = S.fromList [2]
simpleDFA :: DFA' Char
simpleDFA = DFA' alpha states accepting start where
  alpha = S.fromList ['a', 'b', 'c']
  states = M.fromList
    [(0, M.fromList [('a', 1)]),
     (1, M.fromList [('b', 0), ('c', 2)])]
  start = 0
  accepting = S.fromList [2]
deadStateDFA :: DFA' Char
deadStateDFA = DFA' alpha states accepting start where
  alpha = S.fromList "ab"
  states =
    M.fromList [(0, trans0), (1, trans1), (2, trans2)] where
      trans0 = M.fromList [('a', 1), ('b', 2)]
      trans1 = M.fromList [('b', 3)]
      trans2 = M.fromList [('a', 3)]
  accepting = S.fromList [1, 2]
  start = 0

```

3 Regular Expressions

In this module we give the haskell data type for a regular expression; the encoding almost exactly mirrors the definition given in the assignment.

```

module Regex (Regex (..)) where
data Regex a = Alt (Regex a) (Regex a)
  | Concat (Regex a) (Regex a)
  | Repeat (Regex a)
  | Term a
  | Empty deriving Show

```

4 Algorithms

Our solutions to the “in-memory” algorithms given in §1.2 have been modularized in the following way.

```
module Algorithms (module Thompson,  
  module Recognize,  
  module SubsetConstruction) where  
  
  import Thompson  
  import Recognize  
  import SubsetConstruction
```

In this way we encapsulated (and named) the solutions individually, as the assignment requested.

4.1 Thompson’s Algorithm

In this module we provide our solution for converting a regular expression to an NFA.

```
module Thompson (thompson) where  
  
  import Prelude hiding (concat)  
  import qualified Data.Set as S  
  import qualified Data.Map as M  
  import FiniteStateAutomata (FSA (trans), NFA' (..), epsilon)  
  import Regex
```

The function *thompson* returns the result of converting a regular expression to a non-deterministic finite state automaton. It uses Thompson’s algorithm for doing so.

```
thompson :: (Ord a, Show a) => Regex a -> NFA' a  
thompson = fst o thompson' 0  
thompson' :: (Ord a, Show a) =>  
  Int -> Regex a -> (NFA' a, Int)  
thompson' lab (Alt r1 r2) = union lab'' fsa fsa' where  
  (fsa, lab') = thompson' lab r1  
  (fsa', lab'') = thompson' lab' r2  
thompson' lab (Concat r1 r2) = concat lab'' fsa fsa' where
```

```

(fsa, lab') = thompson' lab r1
(fsa', lab'') = thompson' lab' r2
thompson' lab (Repeat r1) = mrKleene lab' fsa where
  (fsa, lab') = thompson' lab r1
thompson' lab (Term x) = symbol lab x
thompson' lab Empty = expression lab

```

The following functions individually convert particular regular expressions to their NFA equivalents. For example, *concat* takes two regular expressions, say *a* and *b*, and returns an NFA where the NFA corresponding to *a*'s accepting states are now transitions to the NFA corresponding to *b*'s start state.

The other functions perform similar operations, according to the algorithm.

```

expression :: (Ord a, Show a) => Int -> (NFA' a, Int)
expression label = (fsa, label + 2) where
  fsa = NFA' S.empty (M.fromList [n1])
  (S.singleton (label + 1)) label
  n1 = (label, S.singleton (epsilon, (label + 1)))
symbol :: (Ord a, Show a) => Int -> a -> (NFA' a, Int)
symbol label sym = (fsa, label + 2) where
  fsa = NFA' (S.singleton sym)
  (uncurry M.singleton n1) (S.singleton (label + 1)) label
  n1 = (label, S.singleton (Just sym, label + 1))
union :: (Ord a, Show a) =>
  Int -> NFA' a -> NFA' a -> (NFA' a, Int)
union label nfa0 nfa1 = (fsa, label + 2) where
  (NFA' a0 m0 as0 st0) = updateAccepting [(label + 1)] nfa0
  (NFA' a1 m1 as1 st1) = updateAccepting [(label + 1)] nfa1
  fsa = NFA' alpha newMap (S.singleton (label + 1)) label
  alpha = S.union a0 a1
  newMap = M.unions [m0, m1, epsilonEdges]
  epsilonEdges =
    M.singleton label
    (S.fromList [(epsilon, st0), (epsilon, st1)])
concat :: (Ord a, Show a) =>
  Int -> NFA' a -> NFA' a -> (NFA' a, Int)

```

```

concat label fsa0@(NFA' s0 m0 as0 st0) (NFA' s1 m1 as1 st1) =
  (fsa, label) where
    fsa = NFA' (S.union s0 s1) (M.union updated m1) as1 st0
    updated = trans $ updateAccepting [st1] fsa0
mrKleene :: (Ord a, Show a) ⇒ Int → NFA' a → (NFA' a, Int)
mrKleene label nfa@(NFA' a _ as st) = (fsa, label + 2) where
  (NFA' _ m _ ) = updateAccepting [st, (label + 1)] nfa
  fsa = NFA' a m' (S.singleton (label + 1)) label
  m' = M.union m epsilons
  epsilons =
    M.singleton label
      (S.fromList [(epsilon, (label + 1)), (epsilon, st)])
  epsilons' = M.fromList ∘ map func ∘ S.toList $ as
  func x = (x, S.singleton (epsilon, label + 1))
updateAccepting :: (Ord a) ⇒ [Int] → NFA' a → NFA' a
updateAccepting is nfa@(NFA' a ts as st) =
  NFA' a newTrans (S.empty) st where
    newTrans = M.union ts nts
    nts = M.fromList ∘ map func ∘ S.toList $ as
    func x =
      (x, S.fromList ∘ map (λi → (epsilon, i)) $ is)

```

4.2 Subset Construction

In this module we provide our solution for converting a given non-deterministic finite state automaton to an equivalent deterministic finite state automaton.

```

{-# LANGUAGE FlexibleInstances #-}
module SubsetConstruction (subsetConstruction) where
import Data.Maybe
import FiniteStateAutomata
import qualified Data.Map as M
import qualified Data.Set as S
type LabelMap = M.Map (S.Set Int) Int
subsetConstruction :: (Ord a, Show a) ⇒ NFA' a → DFA' a
subsetConstruction nfa =
  DFA' (alphabet nfa) dfamap' accept start' where

```



```

start' = labelsM. ! startStateSet
accept = findAccepting nfa labelmap
( _, labelmap, dfamap' ) =
  subsetConstruction' nfa next
  labels dfamap outSets
startStateSet = closure nfa (start nfa)
(labels, next) =
  labelSets 0 M.empty
  (S.fromList (startStateSet : outSets))
edges = edgeMap labels edgeSet
dfamap =
  M.singleton (labelsM. ! startStateSet) edges
outSets =
  map (closure' ◦ flip move' startStateSet)
  alphabet'
edgeSet = zip alphabet' outSets
alphabet' = S.toList ◦ alphabet $ nfa
closure' = closure nfa
move' = move nfa

findAccepting :: (Ord a, Show a) ⇒
  NFA' a → LabelMap → S.Set Int
findAccepting nfa labels = S.fromList sets where
  sets = M.elems (M.filterWithKey isAccepting labels)
  isAccepting label _ =
    S.empty ≢ (S.intersection accept label)
  accept = accepting nfa

subsetConstruction' :: (Ord a, Show a) ⇒
  NFA' a → Int → LabelMap →
  DFAMap a → [S.Set Int] →
  (Int, LabelMap, DFAMap a)
subsetConstruction' _ next labels dfamap [] =
  (next, labels, dfamap)
subsetConstruction' nfa next labels dfamap (s : ss) =
  case (s ≡ S.empty) of
    True → subsetConstruction' nfa next labels dfamap ss
    False → if done then continue else recursion where
      done = M.lookup (labelsM. ! s) dfamap ≢ Nothing

```

```

continue = subsetConstruction' nfa next labels dfamap ss
recursion =
  subsetConstruction' nfa next'' labels'' dfamap'' ss
(next'', labels'', dfamap'') =
  subsetConstruction' nfa next' labels' dfamap' outSets
(labels', next') =
  labelSets next labels (S.fromList outSets)
dfamap' = M.insert (labelsM. ! s) edges dfamap
edges = edgeMap labels' edgeSet
edgeSet = zip alphabet' outSets
outSets = map (closure' o flip move' s) alphabet'
alphabet' = S.toList o alphabet $ nfa
move' = move nfa
closure' = closure nfa

labelSets :: Int → M.Map (S.Set Int) Int →
  S.Set (S.Set Int) →
    (M.Map (S.Set Int) Int, Int)
labelSets next labels sets =
  labelSets' next labels (S.toList sets)
labelSets' :: Int → LabelMap →
  [S.Set Int] → (LabelMap, Int)
labelSets' next labels [] = (labels, next)
labelSets' next labels (s : ss) =
  case (s ≡ S.empty) of
    True → labelSets' next labels ss
    False →
      if (M.member s labels) then
        (labelSets' next labels ss)
      else
        (labelSets' (next + 1)
          (M.insert s next labels) ss)

edgeMap :: (Ord a, Show a) ⇒
  M.Map (S.Set Int) Int →
    [(a, (S.Set Int))] → M.Map a Int
edgeMap = edges' M.empty where
  edges' :: (Ord a, Show a) ⇒
    M.Map a Int → M.Map (S.Set Int) Int →

```

```

    [(a, (S.Set Int))] → M.Map a Int
edges' acc _ [] = acc
edges' acc labels ((alpha, set) : ss) =
  case (set ≡ S.empty) of
    True → edges' acc labels ss
    False → edges' acc' labels ss where
      acc' = M.insert alpha (labelsM. ! set) acc
class Constructable c where
  closure :: (Show a, Ord a) ⇒ NFA' a → c → S.Set Int
  move :: (Show a, Ord a) ⇒ NFA' a → a → c → S.Set Int
instance Constructable Int where
  closure nfa state =
    closure' (S.singleton state) nfa state where
  closure' acc nfa state =
    if done then acc' else acc'' where
    done = edges ≡ Nothing ∨ eps ≡ S.singleton state
    edges = M.lookup state ∘ trans $ nfa
    eps = S.union (S.singleton state)
      (S.map snd ∘ S.filter isEpsilon ∘
        fromJust $ edges)
    eps' = S.difference eps acc
    isEpsilon (label, _) = label ≡ epsilon
    acc' = S.union acc (S.singleton state)
    acc'' = S.unions ∘ S.toList ∘
      S.map (closure' acc' nfa) $ eps'
  move nfa sym state =
    if (edges ≡ Nothing) then S.empty else eps where
    edges = M.lookup state ∘ trans $ nfa
    eps = S.map snd ∘ S.filter isSym ∘ fromJust $ edges
    isSym (label, _) =
      label ≠ Nothing ∧ sym ≡ fromJust label
instance Constructable (S.Set Int) where
  closure nfa states = concatMap' (closure nfa) states
  move nfa sym states = concatMap' (move nfa sym) states
concatMap' :: (Ord a, Ord b) ⇒
  (a → S.Set b) → S.Set a → S.Set b
concatMap' f = S.unions ∘ S.toList ∘ S.map f

```

5 Hopcroft's Algorithm

In this module we provide our solution for minimizing a given deterministic finite state automaton.

The pseudocode for this algorithm from https://en.wikipedia.org/wiki/DFA_minimization is as follows:

```
P := {F, Q \ F};
W := {F};
while (W is not empty) do
  choose and remove a set A from W
  for each c in do
    let X be the set of states
    for which a transition
    on c leads to a state in A
  for each set Y in P for which
  X ∩ Y is nonempty do
    replace Y in P by the
    two sets X ∩ Y and Y \ X
    if Y is in W
      replace Y in W by the same two sets
    else
      if |X ∩ Y| <= |Y \ X|
        add X ∩ Y to W
      else
        add Y \ X to W
  end;
end;
end;
```

```
module Hopcroft (hopcroft) where
import FiniteStateAutomata (FSA (..), DFA' (..))
import Data.Maybe (fromJust, isJust, isNothing)
import qualified Data.Map as M
import qualified Data.Set as S
hopcroft :: (Ord a, Show a) => DFA' a → DFA' a
hopcroft dfa = hopcroft' (dropUnreachable dfa)
```

```

        parts partMap where
    accept'      = accepting dfa
    notAccept    = S.difference (states dfa) accept'
    parts        = S.fromList [accept', notAccept]
    partMap      = toPartitionMap parts
hopcroft' :: (Ord a, Show a) =>
    DFA' a → S.Set (S.Set Int) →
    M.Map Int Int → DFA' a
hopcroft' dfa set eqMap =
    if done then dfa' else recurse where
    done          = consistent' ≡ S.empty
    consistent'   = consistent dfa eqMap ∘ S.toList $ set
    dfa'          = buildDFA dfa eqMap
    recurse       = hopcroft' dfa set' eqMap' where
        set'      = S.union (S.delete consistent' set)
        (partition dfa eqMap consistent')
    eqMap'        = toPartitionMap set'
consistent :: (Ord a, Show a) =>
    DFA' a → M.Map Int Int →
    [S.Set Int] → S.Set Int
consistent _ _ [] = S.empty
consistent dfa eqMap (s:ss') =
    if continue then recurse else s where
    continue = (isConsistent dfa eqMap (S.toList s))
    recurse  = consistent dfa eqMap ss'
buildDFA :: (Ord a, Show a) => DFA' a →
    M.Map Int Int → DFA' a
buildDFA dfa eqMap = DFA' alphabet' ss' accept' st' where
    alphabet'    = alphabet dfa
    ss'           = M.fromList oldStates
    accept'       = S.map lookup' (accepting dfa)
    st'           = lookup' (start dfa)
    newStates     = S.toList ∘ S.fromList ∘ M.elims $ eqMap
    oldStates     = zip newStates ∘
        map (updateState dfa eqMap) ∘ map check $ newStates
    check ns      = M.keys ∘ M.filter (≡ ns) $ eqMap
    lookup'       = (eqMapM!)

```

```

updateState :: (Ord a, Show a) =>
    DFA' a → M.Map Int Int →
    [Int] → M.Map a Int
updateState dfa eqMap oldStates = update where
    update = M.map (eqMapM!) ∘
        M.unions ∘ map fromJust ∘
        filter isJust ∘ map lookup' $ oldStates
    lookup' = flip M.lookup (trans dfa)
    -- Builds a partition map for equivalence look up
toPartitionMap :: S.Set (S.Set Int) → M.Map Int Int
toPartitionMap = toPartitionMap' 0 M.empty ∘ S.toList
    where
    toPartitionMap' _ acc [] = acc
    toPartitionMap' next acc (s:ss') =
        toPartitionMap' (next + 1) acc' ss' where
        acc' = S.fold insert acc s
        insert = flip M.insert next
    -- Partitions a given equivalence group
partition :: (Ord a, Show a) => DFA' a →
    M.Map Int Int → S.Set Int → S.Set (S.Set Int)
partition dfa parts toPart =
    partition' S.empty dfa parts (S.toList toPart)
    where
    partition' acc _ _ [] = acc
    partition' acc dfa parts (s:ss) =
        partition' acc' dfa parts ss' where
        acc' = S.insert set acc
        sMap = eqMap s
        matches = filter ((sMap ≡) ∘ eqMap) ss
        set = S.fromList (s : matches)
        ss' = filter elems ss
        elems x = ¬ (S.member x set)
        eqMap x = eqMap' where
        map' = M.lookup x (trans dfa)
        eqMap' =
            if isNothing map' then
                M.empty

```

```

    else
      equivalenceMap parts ◦ fromJust $ map'
      -- Determines if a set of states all have the same edges
      isConsistent :: (Ord a, Show a) ⇒
        DFA' a → M.Map Int Int → [Int] → Bool
      isConsistent _ _ [] = True
      isConsistent dfa partitions (s : ss) =
        isConsistent' dfa partitions eqMap ss where
          map = M.lookup s (trans dfa)
          eqMap =
            if isNothing map then
              M.empty
            else
              equivalenceMap partitions ◦ fromJust $ map
      equivalenceMap :: M.Map Int Int →
        M.Map a Int → M.Map a Int
      equivalenceMap partitions map' =
        M.mapWithKey updateKey map' where
          updateKey _ v = partitionsM. ! v
      isConsistent' :: (Ord a, Show a) ⇒
        DFA' a → M.Map Int Int →
        M.Map a Int → [Int] → Bool
      isConsistent' _ _ _ [] = True
      isConsistent' dfa partitions eqMap (s : ss') =
        if consistent' then
          recurse
        else
          False where
            consistent' = map' ≡ eqMap
            mMap = M.lookup s (trans dfa)
            map' =
              if isNothing mMap then
                M.empty
              else equivalenceMap partitions ◦ fromJust $ mMap
            recurse = isConsistent' dfa partitions eqMap ss'
      -- Removes all unreachable states in a DFA'
      dropUnreachable :: (Ord a, Show a) ⇒ DFA' a → DFA' a

```

```

dropUnreachable dfa = dropUnreachable' set set dfa where
  set = S.singleton $ start dfa
dropUnreachable' :: (Ord a, Show a) =>
  S.Set Int → S.Set Int → DFA' a → DFA' a
dropUnreachable' reachable_states new_states dfa =
  if done then dfa' else recurse where
    reachable' =
      S.unions ∘ S.toList ∘ S.map (reachable dfa) $ new_states
    new_states' =
      S.difference reachable' reachable_states
    reachable_states' =
      S.union reachable_states new_states'
    recurse =
      dropUnreachable' reachable_states' new_states' dfa
    dfa' =
      updateDFA dfa reachable_states'
    done = new_states' ≡ S.empty
updateDFA :: (Ord a, Show a) =>
  DFA' a → S.Set Int → DFA' a
updateDFA dfa reachable_states =
  DFA' alphabet' trans' accept' start' where
    unreachable_states =
      S.difference (states dfa) reachable_states
    accept' =
      S.difference (accepting dfa) unreachable_states
    alphabet' = alphabet dfa
    start' = start dfa
    trans' = M.filterWithKey removeKey (trans dfa)
    removeKey k _ = S.member k reachable_states
reachable :: (Ord a, Show a) => DFA' a → Int → S.Set Int
reachable fsa state = S.fromList ns where
  trans' = M.lookup state (trans fsa)
  ns = if isNothing trans' then [] else ns'
  ns' = M.elems ∘ fromJust $ trans'
-- A test DFA that has several unreachable states: [3,4,5,6]
testDFA :: DFA' Char
testDFA = DFA' alpha' ss' accept' st' where

```



```

alpha' = S.fromList "ab"
ss'    =
    M.fromList [(0,trans0),
                (1,trans1),
                (2,trans2),
                (3,trans3)]
trans0 = M.fromList [('a',1),('b',2)]
trans1 = M.empty
trans2 = M.empty
trans3 = M.fromList [('a',4),('b',5)]
accept' = S.fromList [1,2,3,6]
st'     = 0

-- Tests the removal of unreachable states
testDroppable :: Bool
testDroppable = alphabet' ∧ states'
                ∧ start' ∧ accepting' where
    alphabet'    = (alphabet dfa) ≡ (S.fromList "ab")
    states'      = (states dfa) ≡ (S.fromList [0,1,2])
    start'       = (start dfa) ≡ 0
    accepting'   = (accepting dfa) ≡ (S.fromList [1,2])
    dfa          = dropUnreachable testDFA

-- A test DFA that can be reduced
-- to a single node with two edges
-- it recognizes strings of the language (a|b)*
testDFA' :: DFA' Char
testDFA' = DFA' alpha' ss' accept' st' where
    alpha' = S.fromList "ab"
    ss'    = M.fromList
        [(0,trans'),
         (1,trans'),
         (2,trans')]
    trans' = M.fromList [('a',1),('b',2)]
    accept' = S.fromList [0,1,2]
    st'     = 0

-- Tests that hopcroft reduces testDFA' to a minimal dfa
testHopcroft :: Bool
testHopcroft = alphabet' ∧ states'

```

```

                                 $\wedge$  accepting'  $\wedge$  trans' where
alphabet'      = (alphabet dfa)  $\equiv$  (S.fromList "ab")
states'        = (states dfa)  $\equiv$  (S.fromList [start'])
start'         = (start dfa)
accepting'     = (accepting dfa)  $\equiv$  (S.fromList [start'])
trans'         = (trans dfa)  $\equiv$  (M.fromList [(start', trans0)])
trans0         = M.fromList [('a', start'), ('b', start')]
dfa            = hopcroft testDFA'

testPartition :: Bool
testPartition = partition'  $\equiv$  correctPartition where
  partition' = partition dfa parts toPart
  correctPartition = S.fromList [s1, s2, s3]
  s1          = S.fromList [1, 2, 5]
  s2          = S.fromList [3]
  s3          = S.fromList [4, 7]
  parts       = M.fromList
    [(0, 0), (6, 0),
     (1, 1), (2, 1),
     (3, 1), (4, 1),
     (5, 1), (7, 1)]
  toPart      = S.fromList [1, 2, 3, 4, 5, 7]
  dfa         = nonMinimalDFA

nonMinimalDFA :: DFA' Char
nonMinimalDFA = DFA' alpha' ss' accept' st' where
  alpha'      = S.fromList "ab"
  ss'         = M.fromList
    [(0, trans0), (1, trans1), (2, trans2),
     (3, trans3), (4, trans4), (5, trans5),
     (6, trans6), (7, trans7)]
  trans0      = M.fromList [('a', 1)]
  trans1      = M.fromList [('a', 4), ('b', 2)]
  trans2      = M.fromList [('a', 3), ('b', 5)]
  trans3      = M.fromList [('b', 1)]
  trans4      = M.fromList [('a', 6), ('b', 5)]
  trans5      = M.fromList [('a', 7), ('b', 2)]
  trans6      = M.fromList [('a', 5)]
  trans7      = M.fromList [('a', 0), ('b', 5)]

```

```

accept' = S.fromList [0,6]
st'      = 0

```

5.1 Recognize a string for a given DFA

In this module we test a string with a given DFA and determine whether the DFA accepts the string or not.

```

module Recognize (match) where
import FiniteStateAutomata
import qualified Data.Map as M
import qualified Data.Set as S

```

The function *match* takes a DFA and a string as input, and returns whether that string is accepted by that DFA. It is fairly straightforward.

```

match :: (Ord a, Show a) => DFA' a -> [a] -> Bool
match dfa = match' dfa (start dfa) where
  match' dfa curr [] = S.member curr (accepting dfa)
  match' dfa curr (c : cs) = let labelMap = M.lookup curr (trans dfa) in
    case labelMap of
      Nothing -> False
      Just map -> let labels = M.lookup c map in
        case labels of
          Nothing -> False
          Just next -> match' dfa next cs

```

6 Alphabet

This module provides functions for lexing and parsing alphabets found in input files, in addition to an alphabet token data structure.

```

module Alphabet where
import Data.List
data Alphabet =
  Symbol Char |
  AlphabetToken |

```

```

    EndToken deriving (Show, Eq)
gotoAlphabet [] = []
gotoAlphabet cs | isPrefixOf "alphabet" cs = cs
gotoAlphabet (c : cs) = gotoAlphabet cs
scanAlphabet [] = []
scanAlphabet ('a' : 'l' : 'p' : 'h' : 'a' : 'b' : 'e' : 't' : cs) =
    AlphabetToken : scanAlphabet cs
scanAlphabet ('\'' : c : cs) =
    Symbol c : scanAlphabet cs
scanAlphabet ('e' : 'n' : 'd' : cs) =
    [EndToken]
scanAlphabet (_ : cs) =
    scanAlphabet cs
parseAlphabet [] = []
parseAlphabet (AlphabetToken : ts) =
    parseAlphabet ts
parseAlphabet (Symbol c : ts) =
    c : parseAlphabet ts
parseAlphabet (EndToken : ts) =
    []
getAlphabet = parseAlphabet ∘ scanAlphabet ∘ gotoAlphabet

```

7 Input

```

module Input (
    module ParseDFA,
    module ParseNFA,
    module ParseReg,
    module ParseLang) where
import ParseDFA
import ParseNFA
import ParseReg
import ParseLang

```

7.1 Parse a Regular Expression

This module inputs, lexes, and parses a regular expression from a text file. It uses Hutton's Parselib library.

Parsing is divided into a function for each regular expression. It handles ascii spaces, newlines, tabs, etc. I.e., the printable subset of ascii, as required by the spec.

```
module ParseReg (getRegex) where
    -- alphabet not being used, need to check for membership
    -- i suppose
import Alphabet
import Regex
import Parselib
parseAlt :: Parser (Regex Char)
parseAlt = do
    string "|"
    space
    regex ← parseRegex'
    space
    regex' ← parseRegex'
    return (Alt regex regex')
parseConcat :: Parser (Regex Char)
parseConcat = do
    string "+"
    space
    regex ← parseRegex'
    space
    regex' ← parseRegex'
    return (Concat regex regex')
parseKleene :: Parser (Regex Char)
parseKleene = do
    string "*"
    space
    regex ← parseRegex'
    return (Repeat regex)
parseTerm :: Parser (Regex Char)
parseTerm = do
```

```

char '\ '
c ← alphanum + + + char ' ' + + + char '\n' + + + char '\t'
return (Term c)
parseRegex' :: Parser (Regex Char)
parseRegex' = do
  space
  parseAlt + + + parseConcat + + + parseKleene + + + parseTerm
getRegex file =
  case (parse parseRegex') file of
    [] → error "Could not parse regular expression."
    regex → (fst ∘ head) regex
-- example
readRegex = do
  source ← readFile "regexp2.txt"
  let regex = getRegex source
  putStrLn $ show regex

```

7.2 Parse an NFA

```

{-# LANGUAGE FlexibleContexts #-}
module ParseNFA where
import Alphabet
import Data.Functor
import FiniteStateAutomata
import Text.Parsec
parseNFA' :: String → NFA' Char
parseNFA' s = ⊥
nfa' :: Stream s m Char ⇒ ParsecT s u m (NFA' Char)
nfa' = do
  string "nfa" >> newline
  string "end;" >> newline
  return ⊥
parseStates :: Stream s m Char ⇒ ParsecT s u m [String]
parseStates = do
  string "states" >> newline
  reverse < $ > parseStates' [] where

```

```

    parseStates' acc = do
      sOrT ← parseStringOrTerm "end;" (many1 (noneOf " \n"))
      newline
      case sOrT of
        Left ter → return acc
        Right st → parseStates' (st : acc)
  parseStringOrTerm :: Stream s m Char ⇒ String →
    ParsecT s u m String →
    ParsecT s u m (Either String String)
  parseStringOrTerm term s = do
    str ← (try $ string term) < | > s
    return $ if str ≡ term then Left str else Right str
  parseAlphabet :: Stream s m Char ⇒ ParsecT s u m [Alphabet]
  parseAlphabet = do
    tok ← AlphabetToken < $ string "alphabet"
    newline
    syms ← sym 'sepBy1' (char ' ')
    newline
    endTok ← EndToken < $ string "end"
    return $ tok : (syms ++ [endTok])
  sym :: Stream s m Char ⇒ ParsecT s u m Alphabet
  sym = char '\\' >> Symbol < $ > anyChar

```

7.3 Parse a DFA

```
module ParseDFA where
```

8 Module: Main.lhs

```

module Main where
import FiniteStateAutomata
import Regex
import Algorithms
import Input

```

```
main = do  
  source ← readFile "regex2.txt"  
  let regex = getRegex source  
  putStrLn $ show regex
```