CS454 Project 1:
« Lexer Generator »

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

March 15, 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.

Lastly, we decided to write a lexer generator for our final output. That is, we parse a lexical description, and return a Haskell source file to be compiled. Once suitably compiled, the binary will accept text files which are accepted if and only if they are strings accepted by the language given in the lexical description.

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 ($\delta : \mathrm{Q} \times \Sigma \to \mathrm{Q}$)

4. a start state (q0 ∈ Q)
5. a set of accept states (F ⊂ Q)

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

```haskell
{-# LANGUAGE TypeFamilies, FlexibleContexts,FlexibleInstances #-}
module FiniteStateAutomata (
                                   FSA (..),
                                   NFA' (..),
                                   NFAMap,
                                   DFA' (..),
                                   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)) => Listable m where
  type Elem m
  toList :: m -> [(Elem m, Int)]
instance (Show a) => Listable (M.Map a Int) where
  type Elem (M.Map a Int) = a
  toList = M.toList
instance (Show a) => 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)) => FSA f where
  type Alpha f
  type FSAVal f
  alphabet :: (Ord (Alpha f), Show (Alpha f)) =>
            f -> S.Set (Alpha f)
  accepting :: f -> S.Set Int
```

```
  start     :: f -> Int
  trans     :: f -> M.Map Int (FSAVal f)
  states    :: f -> S.Set Int
  states fsa = S.unions [(S.fromList . M.keys . 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
```

```haskell
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
```

```haskell
  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.

```haskell
{-# LANGUAGE TypeFamilies, FlexibleContexts,FlexibleInstances #-}
module Regex (Regex (..)) where
data Regex a = Alt (Regex a) (Regex a)
```

```
  | Concat (Regex a) (Regex a)
  | Kleene (Regex a)
  | Term a
  | Empty deriving Show
```

We wrote a pretty infix printer:

```
instance Show (Regex Char) where
    show (Alt r1 r2   ) =
        "(" ++ (show r1) ++ "|" ++ (show r2) ++ ")"
    show (Concat r1 r2) =
        "(" ++ (show r1) ++ "+" ++ (show r2) ++ ")"
    show (Kleene r1)    = "(" ++ (show r1) ++ "*" ++ ")"
    show (Term c)       = '\'':c:[]
    show (Empty)        = "\\epsilon"
```

However, it was determined that writing a lexer generator would be easier if we used the default show for regular expressions.

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 SubsetConstruction,
  module Hopcroft,
  module Recognize
) where
import Thompson
import SubsetConstruction
import Hopcroft
import Recognize
```

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.

$$\textbf{module}\ \mathit{Thompson}\ (\mathit{thompson})\ \textbf{where}$$

$$\begin{array}{l}
\textbf{import}\ \mathit{Prelude\ hiding}\ (\mathit{concat}) \\
\textbf{import}\ \mathit{qualified\ Data.Set\ as\ S} \\
\textbf{import}\ \mathit{qualified\ Data.Map\ as\ M} \\
\textbf{import}\ \mathit{FiniteStateAutomata}\ (\mathit{FSA}\ (\mathit{trans}), \mathit{NFA}'\ (..), \mathit{epsilon}) \\
\textbf{import}\ \mathit{Regex}
\end{array}$$

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.

$$\begin{array}{l}
\mathit{thompson} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Regex}\ a \to \mathit{NFA}'\ a \\
\mathit{thompson} = \mathit{fst} \circ \mathit{thompson}'\ 0
\end{array}$$

$$\begin{array}{l}
\mathit{thompson}' :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
\quad \mathit{Int} \to \mathit{Regex}\ a \to (\mathit{NFA}'\ a, \mathit{Int}) \\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Alt}\ r1\ r2) = \mathit{union}\ \mathit{lab}''\ \mathit{fsa}\ \mathit{fsa}'\ \textbf{where} \\
\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ r1 \\
\quad (\mathit{fsa}', \mathit{lab}'') = \mathit{thompson}'\ \mathit{lab}'\ r2 \\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Concat}\ r1\ r2) = \mathit{concat}\ \mathit{lab}''\ \mathit{fsa}\ \mathit{fsa}'\ \textbf{where} \\
\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ r1 \\
\quad (\mathit{fsa}', \mathit{lab}'') = \mathit{thompson}'\ \mathit{lab}'\ r2 \\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Kleene}\ r1) = \mathit{mrKleene}\ \mathit{lab}'\ \mathit{fsa}\ \textbf{where} \\
\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ r1 \\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Term}\ x) = \mathit{symbol}\ \mathit{lab}\ x \\
\mathit{thompson}'\ \mathit{lab}\ \mathit{Empty} = \mathit{expression}\ \mathit{lab}
\end{array}$$

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.

$$\begin{array}{l}
\mathit{expression} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Int} \to (\mathit{NFA}'\ a, \mathit{Int}) \\
\mathit{expression}\ \mathit{label} = (\mathit{fsa}, \mathit{label} + 2)\ \textbf{where}
\end{array}$$

```
  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
```

```haskell
    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 (labels M.! 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 (labels M.! s) edges dfamap
    edges = edgeMap labels' edgeSet
    edgeSet = zip alphabet' outSets
    outSets = map (closure' ∘ flip move' s) alphabet'
    alphabet' = S.toList ∘ 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.union 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
testNFA :: NFA′ Char
testNFA = NFA′ alpha trans accept st where
  alpha  = S.empty
  accept = S.fromList [0, 1]
  st     = 0
  trans  = M.fromList [(0, trans0), (1, trans1), (2, trans2)] where
    trans0 = S.fromList [(Nothing, 1)]
    trans1 = S.fromList [(Nothing, 2)]
    trans2 = S.fromList [(Nothing, 0)]
```

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;
```

```haskell
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.elems $ 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'
                   ∧ accepting' ∧ trans' where
  alphabet'  = (alphabet dfa) ≡ (S.fromList "ab")
  states'    = (states dfa) ≡ (S.fromList [start'])
  start'     = (start dfa)
  accepting' = (accepting dfa) ≡ (S.fromList [start'])
```

```
  trans′        = (trans dfa) ≡ (M.fromList [(start′, trans0)])
  trans0        = M.fromList [('a', start′), ('b', start′)]
  dfa           = hopcroft testDFA′
testPartition :: Bool
testPartition = partition′ ≡ 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 whether a "string" from an alphabet for a DFA is accepted by that DFA or not.

```
module Recognize (match) where

import FiniteStateAutomata
import qualified Data.Map as M
import qualified Data.Set as S
```

The function *match* takes an α DFA and list of α as input (a "string" in the language of that DFA), and returns true if the string is accepted by that DFA, and false otherwise. 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 (parseElement,
  parseAlphabet,
  getAlphabet,
  gotoGetAlphabet) where

import Data.List
import Parselib
import GHC.Unicode (isPrint)
import Data.Char (ord)
```

A formal description of an alphabet is:

```
Alphabet -> alphabet Elements end;
Elements -> 'Subset_ascii
Elements -> Elements
Subset_ascii ->
            a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|
            w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|
            T|U|V|W|X|Y|Z|\n|\t|\r| |+|-|=|_|)|(|*|&|^|%|$|
            #|@|!|~|`|"|'|;|:|/|?|.|>|,|<|]|[|}|{|||\|0|1|2|
            3|4|5|6|7|8|9|\a|\b|\v|\f|\r
```

The Haskell version of this might be written as follows:

```haskell
data Alphabet =
  AlphabetToken |
  Symbol Char |
  EndToken deriving (Show, Eq)
```

The remaining functions find an alphabet in a file, tokenize, and generate a list of the symbols in the alphabet.

```haskell
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
```

```haskell
parseAlphabet' (EndToken : ts) =
  []
getAlphabet' = parseAlphabet' ∘ scanAlphabet ∘ gotoAlphabet
  -- this was not a fun bug to track down
parseEscapedChar =
     do {string "\\n"; return '\n'}
  +++ do {string "\\t"; return '\t'}
  +++ do {string "\\v"; return '\v'}
  +++ do {string "\\r"; return '\r'}
  +++ do {string "\\b"; return '\b'}
  +++ do {string "\\a"; return '\a'}
  +++ do {string "\\f"; return '\f'}
  +++ do {string "\\"; return '\\'}
printablePlus char =
  let ascii = ord char in
  (ascii ⩾ 7 ∧ ascii ⩽ 13)
   ∨ (ascii ⩾ 32 ∧ ascii ⩽ 126)
parseElement :: Parser Char
parseElement = do
  space
  char '\''
  c ←
    parseEscapedChar +++
    sat printablePlus
  return c
parseAlphabet :: Parser [Char]
parseAlphabet = do
  space
  string "alphabet"
  alphabet ← many parseElement
  space
  string "end;" +++ string "end"  -- need because test file
  return alphabet
getAlphabet file =
  case (parse parseAlphabet) file of
    [] → error "Alphabet is empty (no alphabet provided)."
```

```
    regex → (fst ∘ head) regex
  -- to skip beginning contents and read alphabet
gotoGetAlphabet file =
  case (parse parseAlphabet) (gotoAlphabet file) of
    [] → error "Alphabet is empty (no alphabet provided)."
    regex → (fst ∘ head) regex
```

7 Input

Similar to the Algorithms module, in this module we gather all of our solutions for problems 5-8, which deals with file input/output.

Module *ParseFSA* parses either DFAs or FSAs from text file descriptions; *ParseReg* parses regular expressions from text files (with an alphabet) and tests whether symbols occuring in the regular expression are elements of the alphabet provided in the file; and lastly *ParseLang* provides a data structure for lexical descriptions, reads in a complete lexical description from a text file, and transforms it into our internal representation, for use with the algorithms in the *Algorithms* module.

```
module Input (
  module ParseFSA,
  module ParseReg,
  module ParseLang) where
import ParseFSA
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, parseRegex) where
import Alphabet
```

```haskell
import Regex
import Parselib
type Alphabet = [Char]
parseAlt :: Alphabet -> Parser (Regex Char)
parseAlt alphabet = do
  string "|"
  space
  regex <- parseRegex alphabet
  space
  regex' <- parseRegex alphabet
  return (Alt regex regex')
parseConcat :: Alphabet -> Parser (Regex Char)
parseConcat alphabet = do
  string "+"
  space
  regex <- parseRegex alphabet
  space
  regex' <- parseRegex alphabet
  return (Concat regex regex')
parseKleene :: Alphabet -> Parser (Regex Char)
parseKleene alphabet = do
  string "*"
  space
  regex <- parseRegex alphabet
  return (Kleene regex)
parseTerm :: Alphabet -> Parser (Regex Char)
parseTerm alphabet = do
  c <- parseElement
  if not (elem c alphabet) then
    let msg = "Regular expression contains terminal "
          ++ show c
          ++ " which is not an element of the"
          ++ " alphabet provided." in
        error msg
  else
    return (Term c)
```

```haskell
parseRegex :: Alphabet → Parser (Regex Char)
parseRegex alphabet = do
  space
  parseAlt alphabet +++
    parseConcat alphabet +++
    parseKleene alphabet +++
    parseTerm alphabet
  -- takes an alphabet
getRegex :: String → Alphabet → Regex Char
getRegex file alphabet =
  case (parse (parseRegex alphabet)) file of
    [ ] → error "Could not parse regular expression."
    regex → (fst ∘ head) regex
  -- example, should error
readRegex1 = do
  source ← readFile "regexp3.txt"
    -- get alphabet before, because alphabet is after
  let alphabet = gotoGetAlphabet source
  let regex = getRegex source alphabet
  putStrLn $ show regex
readRegex file = do
  source ← readFile file
  let alphabet = gotoGetAlphabet source
  let regex = getRegex source alphabet
  putStrLn $ show regex
```

7.2 Parse an FSA

In this module we parse a description of an DFA or an NFA and return the appropriate data structure.

Since the formal definition of a lexical description of a language does not contain a description of an NFA or a DFA (only regular expressions), this module was simply used on the provided test cases, and for a basic sanity check on whether our implementation for NFAs and DFAs was correct.

It uses Haskell's Parsec library for parsing.

```haskell
 {-# LANGUAGE FlexibleContexts #-}
module ParseFSA (parseNFA, parseDFA) where
```

```haskell
import Data.Functor
import qualified Data.Set as S
import qualified Data.Map as M
import FiniteStateAutomata
import Text.Parsec

data Transition = NFAT {fromState :: String,
      symbols :: [Char],
      toState :: String}
    | DFAT {fromState :: String,
      symbol :: Char,
      toState :: String}

instance Show Transition where
  show (NFAT f ss t) = "NFAT " ++ f ++ "  " ++ (show ss) ++ " --> " ++ t
  show (DFAT f s t) = "DFAT " ++ f ++ "  " ++ (show s) ++ " --> " ++ t

data Description = Description {states' :: [String],
  startState :: String,
  acceptStates :: [String],
  trans' :: [Transition]} deriving Show

parseNFA :: [Char] -> String -> NFA' Char
parseNFA = parseFSA "nfa" (NFA') (toNFAMap)

parseDFA :: [Char] -> String -> DFA' Char
parseDFA = parseFSA "dfa" (DFA') (toDFAMap)

parseFSA typ constr toMap alpha s =
  case parse (description typ isNFA) "Syntax Error" s of
    Left er -> error . show $ er
    Right desc -> convertToFSA alpha desc constr toMap
  where isNFA = if typ == "nfa" then True else False

convertToFSA :: FSA f => [Alpha f] -> Description ->
  (S.Set (Alpha f) -> M.Map Int (FSAVal f) -> S.Set Int -> Int -> f) ->
  (M.Map String Int -> [Transition] -> M.Map Int (FSAVal f)) ->
  f
convertToFSA alpha desc const toMap = const alphabet nfaMap accpting start where
  normal = M.fromList $ zip (states' desc) [0 ..]
  alphabet = S.fromList alpha
  nfaMap = (toMap normal) . trans' $ desc
  accpting = S.fromList . map (normalM.!) . acceptStates $ desc
```

```haskell
  start = normalM ! (startState desc)

toNFAMap :: M.Map String Int -> [Transition] -> NFAMap Char
toNFAMap m ts = M.fromList . map convert . M.toList . go ts $ M.empty where
  convert (s, es) = (m M.! s, S.fromList . map (\(c, s2) -> (c, m M.! s2)) $ es)
  toM "" = [Nothing]
  toM s = map Just s
  go [] acc = acc
  go ((NFAT f syms t) : ts) acc = case M.lookup f acc of
    Nothing -> go ts $ M.insert f (zip (toM syms) (repeat t)) acc
    Just es -> go ts $ M.insert f ((zip (toM syms) (repeat t)) ++ es) acc

toDFAMap :: M.Map String Int -> [Transition] -> DFAMap Char
toDFAMap m ts = M.fromList . map convert . M.toList . go ts $ M.empty where
  convert (s, es) = (m M.! s, M.fromList . map (\(c, s2) -> (c, m M.! s2)) $ es)
  go [] acc = acc
  go ((DFAT f symb t) : ts) acc = case M.lookup f acc of
    Nothing -> go ts $ M.insert f [(symb, t)] acc
    Just es -> go ts $ M.insert f ((symb, t) : es) acc

description :: Stream s m Char =>
        String ->
        Bool ->
        ParsecT s u m Description
description keyword isNFA = do
  spaces >> string keyword >> spaces
  stats <- statelist "states" "end;" identifier
  initState <- initialState "initial"
  acceptStates <- statelist "accept" "end;" identifier
  trans <- statelist "transitions" "end;" (transition isNFA)
  return $ Description stats initState acceptStates trans

transition :: Stream s m Char =>
  Bool ->
  ParsecT s u m Transition
transition isNFA = do
  from <- identifier
  syms <- option [] symbolList
  string "-->" >> spaces
  to <- identifier
  return $ case isNFA of
```

```haskell
    True -> NFAT from syms to
    False -> DFAT from (head syms) to
initialState :: Stream s m Char => String -> ParsecT s u m String
initialState keyword = string keyword >> spaces >> identifier
parseStringOrTerm :: Stream s m Char => String ->
  ParsecT s u m a ->
  ParsecT s u m (Either String a)
parseStringOrTerm term s = do
  ter <- try $ optionMaybe $ string term
  case ter of
    Just t -> return $ Left t
    Nothing -> Right <$> s
statelist :: Stream s m Char =>
  String ->
  String ->
  ParsecT s u m a ->
  ParsecT s u m [a]
statelist startTok endTok elem = do
  string startTok >> spaces
  reverse <$> parseSets' [] where
    parseSets' acc = do
      sOrT <- parseStringOrTerm endTok elem
      spaces
      case sOrT of
        Left _ -> return acc
        Right str -> parseSets' $ str : acc
identifier :: Stream s m Char => ParsecT s u m String
identifier = do
  i <- many1 alphaNum
  spaces
  return i
sym :: Stream s m Char => ParsecT s u m Char
sym = do
  char '\''
  c <- anyChar
  case c of
    '\\' -> do
```

```
        c2 ← anyChar
        return $ read $ "'\\" ++ [c2] ++ "'"
      _ → return c
symbolList :: Stream s m Char ⇒ ParsecT s u m [Char]
symbolList = sym `sepEndBy1` (spaces)
```

7.3 Parse a Lexical Description of a Programming Language

In this module we parse a lexical description of a language, and prepare it for parsing with respect to our previous data structures and algorithms.

It also uses Hutton's Parselib.

```
{-# LANGUAGE TypeFamilies, FlexibleContexts,FlexibleInstances #-}
module ParseLang where
import Parselib
import Regex
import ParseReg
import FiniteStateAutomata
import Alphabet
import Data.Char (isSpace)
import Data.List (intersperse)
```

We use a data structure *Desc* to internally represent a lexical description. *Desc* is a basic record type, with three functions, *language*, *symbols*, and *classes* which return the name of the language as a string, the alphabet, and a list of classes given by the lexical description, respectively.

The data structure *Class* is another record type with three functions, *name*, *regex*, and *relevance*, which return the name of the class, the regular expression which describes it, and its semantic relevance, respectively.

Thus to obtain obtain an NFA equivalent of the regular expression for the first class given in a parsed lexical description l, we write: $(thompson \circ regex \circ head \circ classes)\ l$.

```
type Identifier = String
data Relevance = Relevant | Irrelevant | Discard
instance Show Relevance where
  show (Relevant) = "relevant"
```

```
  show (Irrelevant) = "irrelevant"
  show (Discard) = "discard"
data Class = Class {
  name :: Identifier,
  regex :: Regex Char,
  relevance :: Relevance}
instance Show Class where
  show c = "class " ++ name c ++ " " ++
    show (regex c) ++ " " ++
    show (relevance c) ++
    " end;"
data Desc = Desc {
  language :: String,
  symbols :: [Char],
  classes :: [Class]
  }
showAlphabet a = "'" ++ (intersperse '\'' a)
instance Show Desc where
  show desc =
    "language: " ++ language desc ++ "\n" ++
    "alphabet: " ++
    showAlphabet (symbols desc) ++
    " end;" ++ "\n" ++
    "classes: " ++ "\n" ++
    unlines (map show (classes desc)) ++
    " end;"
```

The remaining functions parse a text file of a lexical description, and deposit that description (if it is well-formed), into our data structure.

```
parseLangIdentifier :: Parser String
parseLangIdentifier = do
  ident ← many $ sat (¬ ∘ isSpace)
  return ident
parseRelevance :: Parser Relevance
parseRelevance =
    do {string "relevant"; return Relevant} +++
    do {string "irrelevant"; return Irrelevant} +++
```

```
    do {string "discard" ;return Discard}
parseClass :: [Char] → Parser Class
parseClass alphabet = do
  space
  string "class"
  space
  name ← parseLangIdentifier
  space
  string "is"
  regex ← parseRegex alphabet
  space
  relevance ← parseRelevance
  space
  string "end;"
  return $ Class name regex relevance
parseLang :: Parser Desc
parseLang = do
  space
  string "language"
  space
  language ← parseLangIdentifier
  alphabet ← parseAlphabet
  classes ← many $ parseClass alphabet
  space
  string "end;"
  return (Desc language alphabet classes)
getLang :: FilePath → IO Desc
getLang file = do
  source ← readFile file
  case (parse parseLang) source of
    [ ] → error "Could not parse lexical description."
    regex → return $ (fst ∘ head) regex
  -- example
readLang1 = do
  source ← readFile "tests/lexdesc3.txt"
  let x = (parse parseLang) source
  putStrLn $ show x
```

```
readLang file = do
  source ← readFile file
  let x = (parse parseLang) source
  putStrLn $ show x
```

8 Module: Main.lhs

The final module, *Main*, puts everything together.

```
module Main where
import FiniteStateAutomata
import Regex
import Algorithms
import Input
alternate (c : [ ]) = regex c
alternate (c : cs) =
  Alt (regex c) (alternate cs)
```

From a given lexical description, we first alternate all of the regular expressions found in the classes, then kleene star the entire expression; then we apply Thompson's algorithm, then generate a dfa from the nfa, then apply Hopcroft's minimization algorithm, then finally check whether the dfa recognizes a given string.

```
main = do
  testfile ← readFile "tests/testfile3.txt"
  desc ← getLang "tests/lexdesc3.txt"
  let regex = Kleene (alternate (classes desc))
  let test = ((match ∘ hopcroft ∘ subsetConstruction ∘ thompson) regex) testfile
  putStrLn $ show test
  putStrLn $ show desc
```