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 (δ : Q $\times$ Σ $\rightarrow$ Q)
4. a start state (q0 $\in$ 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.

```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
ppfsa :: (FSA f) ⇒ f → IO ()
ppfsa = pettyPrinter
showTransitions :: (FSA f) ⇒ f → [String]
showTransitions fsa = map showTransition ∘
   M.toList ∘ trans $ fsa where
```

```haskell
  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.

$$\begin{array}{l}
\textbf{module}\ \mathit{Algorithms}\ (\textbf{module}\ \mathit{Thompson},\\
\quad \textbf{module}\ \mathit{Recognize},\\
\quad \textbf{module}\ \mathit{SubsetConstruction})\ \textbf{where}\\
\textbf{import}\ \mathit{Thompson}\\
\textbf{import}\ \mathit{Recognize}\\
\textbf{import}\ \mathit{SubsetConstruction}
\end{array}$$

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.

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

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.

$$\begin{array}{l}
\mathit{thompson} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Regex}\ a \to \mathit{NFA}'\ a\\
\mathit{thompson} = \mathit{fst} \circ \mathit{thompson}'\ 0\\
\mathit{thompson}' :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow\\
\quad \mathit{Int} \to \mathit{Regex}\ a \to (\mathit{NFA}'\ a, \mathit{Int})\\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Alt}\ r1\ r2) = \mathit{union}\ \mathit{lab}''\ \mathit{fsa}\ \mathit{fsa}'\ \textbf{where}\\
\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ r1\\
\quad (\mathit{fsa}', \mathit{lab}'') = \mathit{thompson}'\ \mathit{lab}'\ r2\\
\mathit{thompson}'\ \mathit{lab}\ (\mathit{Concat}\ r1\ r2) = \mathit{concat}\ \mathit{lab}''\ \mathit{fsa}\ \mathit{fsa}'\ \textbf{where}
\end{array}$$

$$
\begin{aligned}
&\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ \mathit{r1} \\
&\quad (\mathit{fsa}', \mathit{lab}'') = \mathit{thompson}'\ \mathit{lab}'\ \mathit{r2} \\
&\mathit{thompson}'\ \mathit{lab}\ (\mathit{Repeat}\ \mathit{r1}) = \mathit{mrKleene}\ \mathit{lab}'\ \mathit{fsa}\ \mathbf{where} \\
&\quad (\mathit{fsa}, \mathit{lab}') = \mathit{thompson}'\ \mathit{lab}\ \mathit{r1} \\
&\mathit{thompson}'\ \mathit{lab}\ (\mathit{Term}\ x) = \mathit{symbol}\ \mathit{lab}\ x \\
&\mathit{thompson}'\ \mathit{lab}\ \mathit{Empty} = \mathit{expression}\ \mathit{lab}
\end{aligned}
$$

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{aligned}
&\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)\ \mathbf{where} \\
&\quad \mathit{fsa} = \mathit{NFA}'\ \mathit{S.empty}\ (\mathit{M.fromList}\ [\mathit{n1}]) \\
&\qquad (\mathit{S.singleton}\ (\mathit{label} + 1))\ \mathit{label} \\
&\quad \mathit{n1} = (\mathit{label}, \mathit{S.singleton}\ (\mathit{epsilon}, (\mathit{label} + 1))) \\
&\mathit{symbol} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Int} \to a \to (\mathit{NFA}'\ a, \mathit{Int}) \\
&\mathit{symbol}\ \mathit{label}\ \mathit{sym} = (\mathit{fsa}, \mathit{label} + 2)\ \mathbf{where} \\
&\quad \mathit{fsa} = \mathit{NFA}'\ (\mathit{S.singleton}\ \mathit{sym}) \\
&\qquad (\mathit{uncurry}\ \mathit{M.singleton}\ \mathit{n1})\ (\mathit{S.singleton}\ (\mathit{label} + 1))\ \mathit{label} \\
&\quad \mathit{n1} = (\mathit{label}, \mathit{S.singleton}\ (\mathit{Just}\ \mathit{sym}, \mathit{label} + 1)) \\
&\mathit{union} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
&\quad \mathit{Int} \to \mathit{NFA}'\ a \to \mathit{NFA}'\ a \to (\mathit{NFA}'\ a, \mathit{Int}) \\
&\mathit{union}\ \mathit{label}\ \mathit{nfa0}\ \mathit{nfa1} = (\mathit{fsa}, \mathit{label} + 2)\ \mathbf{where} \\
&\quad (\mathit{NFA}'\ \mathit{a0}\ \mathit{m0}\ \mathit{as0}\ \mathit{st0}) = \mathit{updateAccepting}\ [(\mathit{label} + 1)]\ \mathit{nfa0} \\
&\quad (\mathit{NFA}'\ \mathit{a1}\ \mathit{m1}\ \mathit{as1}\ \mathit{st1}) = \mathit{updateAccepting}\ [(\mathit{label} + 1)]\ \mathit{nfa1} \\
&\quad \mathit{fsa} = \mathit{NFA}'\ \mathit{alpha}\ \mathit{newMap}\ (\mathit{S.singleton}\ (\mathit{label} + 1))\ \mathit{label} \\
&\quad \mathit{alpha} = \mathit{S.union}\ \mathit{a0}\ \mathit{a1} \\
&\quad \mathit{newMap} = \mathit{M.unions}\ [\mathit{m0}, \mathit{m1}, \mathit{epsilonEdges}] \\
&\quad \mathit{epsilonEdges} = \\
&\qquad \mathit{M.singleton}\ \mathit{label} \\
&\qquad\quad (\mathit{S.fromList}\ [(\mathit{epsilon}, \mathit{st0}), (\mathit{epsilon}, \mathit{st1})]) \\
&\mathit{concat} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
&\quad \mathit{Int} \to \mathit{NFA}'\ a \to \mathit{NFA}'\ a \to (\mathit{NFA}'\ a, \mathit{Int})
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{concat}\ \mathit{label}\ \mathit{fsa0}@(\mathit{NFA}'\ \mathit{s0}\ \mathit{m0}\ \mathit{as0}\ \mathit{st0})\ (\mathit{NFA}'\ \mathit{s1}\ \mathit{m1}\ \mathit{as1}\ \mathit{st1}) = \\
\quad (\mathit{fsa}, \mathit{label})\ \mathbf{where} \\
\quad \mathit{fsa} = \mathit{NFA}'\ (\mathit{S.union}\ \mathit{s0}\ \mathit{s1})\ (\mathit{M.union}\ \mathit{updated}\ \mathit{m1})\ \mathit{as1}\ \mathit{st0} \\
\quad \mathit{updated} = \mathit{trans}\ \$\ \mathit{updateAccepting}\ [\mathit{st1}]\ \mathit{fsa0} \\
\mathit{mrKleene} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{Int} \rightarrow \mathit{NFA}'\ a \rightarrow (\mathit{NFA}'\ a, \mathit{Int}) \\
\mathit{mrKleene}\ \mathit{label}\ \mathit{nfa}@(\mathit{NFA}'\ a\ \_\ \mathit{as}\ \mathit{st}) = (\mathit{fsa}, \mathit{label} + 2)\ \mathbf{where} \\
\quad (\mathit{NFA}'\ \_\ m\ \_\ \_) = \mathit{updateAccepting}\ [\mathit{st}, (\mathit{label} + 1)]\ \mathit{nfa} \\
\quad \mathit{fsa} = \mathit{NFA}'\ a\ m'\ (\mathit{S.singleton}\ (\mathit{label} + 1))\ \mathit{label} \\
\quad m' = \mathit{M.union}\ m\ \mathit{epsilons} \\
\quad \mathit{epsilons} = \\
\qquad \mathit{M.singleton}\ \mathit{label} \\
\qquad\quad (\mathit{S.fromList}\ [(\mathit{epsilon}, (\mathit{label} + 1)), (\mathit{epsilon}, \mathit{st})]) \\
\quad \mathit{epsilons}' = \mathit{M.fromList} \circ \mathit{map}\ \mathit{func} \circ \mathit{S.toList}\ \$\ \mathit{as} \\
\quad \mathit{func}\ x = (x, \mathit{S.singleton}\ (\mathit{epsilon}, \mathit{label} + 1)) \\
\mathit{updateAccepting} :: (\mathit{Ord}\ a) \Rightarrow [\mathit{Int}] \rightarrow \mathit{NFA}'\ a \rightarrow \mathit{NFA}'\ a \\
\mathit{updateAccepting}\ \mathit{is}\ \mathit{nfa}@(\mathit{NFA}'\ a\ \mathit{ts}\ \mathit{as}\ \mathit{st}) = \\
\quad \mathit{NFA}'\ a\ \mathit{newTrans}\ (\mathit{S.empty})\ \mathit{st}\ \mathbf{where} \\
\qquad \mathit{newTrans} = \mathit{M.union}\ \mathit{ts}\ \mathit{nts} \\
\qquad \mathit{nts} = \mathit{M.fromList} \circ \mathit{map}\ \mathit{func} \circ \mathit{S.toList}\ \$\ \mathit{as} \\
\qquad \mathit{func}\ x = \\
\qquad\quad (x, \mathit{S.fromList} \circ \mathit{map}\ (\lambda i \rightarrow (\mathit{epsilon}, i))\ \$\ \mathit{is})
\end{array}
$$

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.

$$
\begin{array}{l}
\text{\{-\# LANGUAGE FlexibleInstances \#-\}} \\
\mathbf{module}\ \mathit{SubsetConstruction}\ (\mathit{subsetConstruction})\ \mathbf{where} \\
\mathbf{import}\ \mathit{Data.Maybe} \\
\mathbf{import}\ \mathit{FiniteStateAutomata} \\
\mathbf{import}\ \mathit{qualified}\ \mathit{Data.Map}\ \mathit{as}\ M \\
\mathbf{import}\ \mathit{qualified}\ \mathit{Data.Set}\ \mathit{as}\ S \\
\mathbf{type}\ \mathit{LabelMap} = \mathit{M.Map}\ (\mathit{S.Set}\ \mathit{Int})\ \mathit{Int} \\
\mathit{subsetConstruction} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \mathit{NFA}'\ a \rightarrow \mathit{DFA}'\ a \\
\mathit{subsetConstruction}\ \mathit{nfa} = \\
\qquad \mathit{DFA}'\ (\mathit{alphabet}\ \mathit{nfa})\ \mathit{dfamap}'\ \mathit{accept}\ \mathit{start}'\ \mathbf{where}
\end{array}
$$

$$
\begin{array}{l}
\qquad\quad \mathit{start}' = \mathit{labelsM}.\,!\,\mathit{startStateSet} \\
\qquad\quad \mathit{accept} = \mathit{findAccepting}\ \mathit{nfa}\ \mathit{labelmap} \\
\qquad\quad (\_, \mathit{labelmap}, \mathit{dfamap}') = \\
\qquad\qquad \mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next} \\
\qquad\qquad\quad \mathit{labels}\ \mathit{dfamap}\ \mathit{outSets} \\
\qquad\quad \mathit{startStateSet} = \mathit{closure}\ \mathit{nfa}\ (\mathit{start}\ \mathit{nfa}) \\
\qquad\quad (\mathit{labels}, \mathit{next}) = \\
\qquad\qquad \mathit{labelSets}\ 0\ \mathit{M.empty} \\
\qquad\qquad (\mathit{S.fromList}\ (\mathit{startStateSet} : \mathit{outSets})) \\
\qquad\quad \mathit{edges} = \mathit{edgeMap}\ \mathit{labels}\ \mathit{edgeSet} \\
\qquad\quad \mathit{dfamap} = \\
\qquad\qquad \mathit{M.singleton}\ (\mathit{labelsM}.\,!\,\mathit{startStateSet})\ \mathit{edges} \\
\qquad\quad \mathit{outSets} = \\
\qquad\qquad \mathit{map}\ (\mathit{closure}' \circ \mathit{flip}\ \mathit{move}'\ \mathit{startStateSet}) \\
\qquad\qquad\quad \mathit{alphabet}' \\
\qquad\quad \mathit{edgeSet} = \mathit{zip}\ \mathit{alphabet}'\ \mathit{outSets} \\
\qquad\quad \mathit{alphabet}' = \mathit{S.toList} \circ \mathit{alphabet}\ \$\ \mathit{nfa} \\
\qquad\quad \mathit{closure}' = \mathit{closure}\ \mathit{nfa} \\
\qquad\quad \mathit{move}' = \mathit{move}\ \mathit{nfa} \\
\mathit{findAccepting} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
\quad \mathit{NFA}'\ a \rightarrow \mathit{LabelMap} \rightarrow \mathit{S.Set}\ \mathit{Int} \\
\mathit{findAccepting}\ \mathit{nfa}\ \mathit{labels} = \mathit{S.fromList}\ \mathit{sets}\ \mathbf{where} \\
\quad \mathit{sets} = \mathit{M.elems}\ (\mathit{M.filterWithKey}\ \mathit{isAccepting}\ \mathit{labels}) \\
\quad \mathit{isAccepting}\ \mathit{label}\ \_ = \\
\qquad\quad \mathit{S.empty} \not\equiv (\mathit{S.intersection}\ \mathit{accept}\ \mathit{label}) \\
\quad \mathit{accept} = \mathit{accepting}\ \mathit{nfa} \\
\mathit{subsetConstruction}' :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
\quad \mathit{NFA}'\ a \rightarrow \mathit{Int} \rightarrow \mathit{LabelMap} \rightarrow \\
\quad \mathit{DFAMap}\ a \rightarrow [\mathit{S.Set}\ \mathit{Int}] \rightarrow \\
\quad (\mathit{Int}, \mathit{LabelMap}, \mathit{DFAMap}\ a) \\
\mathit{subsetConstruction}'\ \_\ \mathit{next}\ \mathit{labels}\ \mathit{dfamap}\ [\,] = \\
\qquad (\mathit{next}, \mathit{labels}, \mathit{dfamap}) \\
\mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next}\ \mathit{labels}\ \mathit{dfamap}\ (s : \mathit{ss}) = \\
\qquad \mathbf{case}\ (s \equiv \mathit{S.empty})\ \mathbf{of} \\
\quad \mathit{True} \rightarrow \mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next}\ \mathit{labels}\ \mathit{dfamap}\ \mathit{ss} \\
\quad \mathit{False} \rightarrow \mathbf{if}\ \mathit{done}\ \mathbf{then}\ \mathit{continue}\ \mathbf{else}\ \mathit{recursion}\ \mathbf{where} \\
\qquad \mathit{done} = \mathit{M.lookup}\ (\mathit{labelsM}.\,!\,s)\ \mathit{dfamap} \not\equiv \mathit{Nothing}
\end{array}
$$

$$
\begin{array}{l}
\quad\quad \mathit{continue} = \mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next}\ \mathit{labels}\ \mathit{dfamap}\ \mathit{ss} \\
\quad\quad \mathit{recursion} = \\
\quad\quad\quad \mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next}''\ \mathit{labels}''\ \mathit{dfamap}''\ \mathit{ss} \\
\quad\quad (\mathit{next}'', \mathit{labels}'', \mathit{dfamap}'') = \\
\quad\quad\quad \mathit{subsetConstruction}'\ \mathit{nfa}\ \mathit{next}'\ \mathit{labels}'\ \mathit{dfamap}'\ \mathit{outSets} \\
\quad\quad (\mathit{labels}', \mathit{next}') = \\
\quad\quad\quad \mathit{labelSets}\ \mathit{next}\ \mathit{labels}\ (\mathit{S.fromList}\ \mathit{outSets}) \\
\quad\quad \mathit{dfamap}' = \mathit{M.insert}\ (\mathit{labelsM.}\,!\,s)\ \mathit{edges}\ \mathit{dfamap} \\
\quad\quad \mathit{edges} = \mathit{edgeMap}\ \mathit{labels}'\ \mathit{edgeSet} \\
\quad\quad \mathit{edgeSet} = \mathit{zip}\ \mathit{alphabet}'\ \mathit{outSets} \\
\quad\quad \mathit{outSets} = \mathit{map}\ (\mathit{closure}' \circ \mathit{flip}\ \mathit{move}'\ s)\ \mathit{alphabet}' \\
\quad\quad \mathit{alphabet}' = \mathit{S.toList} \circ \mathit{alphabet}\ \$\ \mathit{nfa} \\
\quad\quad \mathit{move}' = \mathit{move}\ \mathit{nfa} \\
\quad\quad \mathit{closure}' = \mathit{closure}\ \mathit{nfa} \\
\mathit{labelSets} :: \mathit{Int} \to \mathit{M.Map}\ (\mathit{S.Set}\ \mathit{Int})\ \mathit{Int} \to \\
\quad \mathit{S.Set}\ (\mathit{S.Set}\ \mathit{Int}) \to \\
\quad\quad\quad (\mathit{M.Map}\ (\mathit{S.Set}\ \mathit{Int})\ \mathit{Int}, \mathit{Int}) \\
\mathit{labelSets}\ \mathit{next}\ \mathit{labels}\ \mathit{sets} = \\
\quad\quad \mathit{labelSets}'\ \mathit{next}\ \mathit{labels}\ (\mathit{S.toList}\ \mathit{sets}) \\
\mathit{labelSets}' :: \mathit{Int} \to \mathit{LabelMap} \to \\
\quad\quad\quad\quad [\mathit{S.Set}\ \mathit{Int}] \to (\mathit{LabelMap}, \mathit{Int}) \\
\mathit{labelSets}'\ \mathit{next}\ \mathit{labels}\ [\,] = (\mathit{labels}, \mathit{next}) \\
\mathit{labelSets}'\ \mathit{next}\ \mathit{labels}\ (s : \mathit{ss}) = \\
\quad \mathbf{case}\ (s \equiv \mathit{S.empty})\ \mathbf{of} \\
\quad\quad \mathit{True} \to \mathit{labelSets}'\ \mathit{next}\ \mathit{labels}\ \mathit{ss} \\
\quad\quad \mathit{False} \to \\
\quad\quad\quad \mathbf{if}\ (\mathit{M.member}\ s\ \mathit{labels})\ \mathbf{then} \\
\quad\quad\quad\quad (\mathit{labelSets}'\ \mathit{next}\ \mathit{labels}\ \mathit{ss}) \\
\quad\quad\quad \mathbf{else} \\
\quad\quad\quad\quad (\mathit{labelSets}'\ (\mathit{next} + 1) \\
\quad\quad\quad\quad\quad (\mathit{M.insert}\ s\ \mathit{next}\ \mathit{labels})\ \mathit{ss}) \\
\mathit{edgeMap} :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
\quad\quad\quad \mathit{M.Map}\ (\mathit{S.Set}\ \mathit{Int})\ \mathit{Int} \to \\
\quad\quad\quad\quad [(a, (\mathit{S.Set}\ \mathit{Int}))] \to \mathit{M.Map}\ a\ \mathit{Int} \\
\mathit{edgeMap} = \mathit{edges}'\ \mathit{M.empty}\ \mathbf{where} \\
\quad \mathit{edges}' :: (\mathit{Ord}\ a, \mathit{Show}\ a) \Rightarrow \\
\quad\quad\quad\quad \mathit{M.Map}\ a\ \mathit{Int} \to \mathit{M.Map}\ (\mathit{S.Set}\ \mathit{Int})\ \mathit{Int} \to
\end{array}
$$

```
        [(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;
```

```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
```

```haskell
          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 = partitions M.! 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)]
```

```haskell
    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.

```haskell
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.

```haskell
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.

```haskell
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
```

```haskell
main = do
  source <- readFile "regexp2.txt"
  let regex = getRegex source
  putStrLn $ show regex
```