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 ($: Q \times \Sigma \to Q$)
4. a start state (q0 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,
                              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.union (S.fromList ∘ M.keys $ (trans fsa))
             (accepting fsa)
```

```
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,
```

```haskell
  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]
  -- NEW ----------------------------------
data DFA a = DFA {q :: [Int],
        sigma :: [a],
        delta :: M.Map (Int, a) Int,
        q0 :: Int,
        f :: [Int]
        } deriving Show
data Trans = Epsilon | Q Int deriving Show
data NFA a = NFA {nq :: [Int],
        nsigma :: [a],
        ndelta :: M.Map (Trans, a) Int,
        nq0 :: Int,
        nf :: [Int]
        } deriving Show
dfa1 = DFA
  [0,1,2,3,4,5,6,7]
  ["a","b"]
  d
  0
  [0,6]
  where
  d = M.fromList [
      ((0,"a"),1),
      ((1,"a"),4),
      ((1,"b"),2),
      ((2,"a"),3),
      ((2,"b"),5),
      ((3,"b"),1),
      ((4,"a"),6),
      ((4,"b"),5),
```

```
((5,"a"),7),
((5,"b"),2),
((6,"a"),5),
((7,"b"),5)]
```

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
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.

```haskell
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.

```haskell
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
thompson :: (Ord a, Show a) ⇒ Regex a → NFA′ a
thompson = fst ∘ 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
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 (\i -> (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] →
subsetConstruction′ nfa 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′ ∘ 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.in
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
  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 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
```

6 Input

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

import ParseDFA
import ParseNFA
```

```
import ParseReg
import ParseLang
```

6.1 Parse a Regular Expression

This module inputs, lexes, and parses a regular expression from a text file.

```
module ParseReg where
  -- alphabet not being used, need to check for membership
  -- i suppose
import Alphabet
import Regex
import Data.List
import Data.Char (isSpace)
import Parselib
data Tokens =
  AltToken |
  ConcatToken |
  KleeneToken |
  TermToken Char |
  EmptyToken deriving (Show, Eq)
tokenize [] = []
tokenize ('|' : cs) = AltToken : tokenize cs
tokenize ('+' : cs) = ConcatToken : tokenize cs
tokenize ('*' : cs) = KleeneToken : tokenize cs
tokenize ('\'' : ' ' : cs) = EmptyToken : tokenize cs
tokenize ('\'' : c : cs) = TermToken c : tokenize cs
tokenize (cs) | isPrefixOf "alphabet" cs = []
tokenize (c : cs) | isSpace c = tokenize cs
tokenize (c : cs) = error ("unknown character"
   ++ show c ++ " in regular expression")
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' = parse parseRegex'
  -- example
readRegex = do
  source ← readFile "regexp2.txt"
  let regex = getRegex' source
  putStrLn $ show regex
```

7 Module: Main.lhs

```
module Main where
```

```
import FiniteStateAutomata
import Regex
import Algorithms
import Input
main =
  putStrLn "(( .x x) helloworld)"
```