Advanced Programming
Andrew Black and Tim Sheard

Lecture 9
More Regular Expressions
NFAs
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

val rel = Concat (Union (C '+')
  (Union (C '-')
    Empty))
  (Concat (C 'D')
    (Star (C 'D'))))
alpha = Union (C 'a')
    (Union (C 'b') (C 'c'))

digit = Union (C '0')
    (Union (C '1') (C '2'))

key = Union (string "if")
    (Union (string "then")
        (string "else"))

punc = (C ',',)

ident = Concat alpha
    (Star (Union alpha digit))

number = Concat digit (Star digit)

lexer = Union ident
    (Union number (Union punc key))
In Assignment #6 we asked for

\texttt{match1} :: \texttt{RE} \to \texttt{Char} \to \texttt{Bool}

(match1 x c) returns True if c is the first character of a
string generated by x. This should be an exact answer
and not limit (Star x) to three iterations.

\texttt{consumes1} :: \texttt{RE} \to \texttt{Char} \to \texttt{Maybe RE}

(consumes1 x c) returns (Just y) if (c:cs) is a string
generated by x, and cs is a string generated by y. Again,
this should be an exact answer and not limit (Star x) to
three iterations.
A non-approximate recognizer

• By repeated use of consumes1 we can construct our first non-approximate recognizer

```haskell
matchA "" Empty = True
matchA ""  r = error
        ("end of string, but left to match: "++show r)
matchA (x:xs) r =
    case consumes1 r x of
        Nothing -> False
        Just r2 -> matchA xs r2
```
Testing

Main> matchA "then" punc
False

Main> matchA "then" key
True

Main> matchA "tom" key
False

Main> matchA "ab2" ident
Program error: end of string,
    but left to matchA: (a+b+c+0+1+2)*
what went wrong here?
matchB "" Empty = True
matchB "" (Star _) = True
matchB "" r = error
    ("end of string, but left to matchB: "++show r)
matchB (x:xs) r =
    case consumes1 r x of
      Nothing -> False
      Just r2 -> matchB xs r2

Main> matchB "ab1c2aa" ident
True

Main> matchB "ab" (Concat (string "ab")
                      (Union (Star (C '1')) Empty))
Program error: end of string, but left to matchB: (1*+#)
Finally

matchC "" r =
  if nullable r
  then True
  else error
      ("end of string, but left to matchC: "++show r)
matchC (x:xs) r =
  case consumes1 r x of
      Nothing    -> False
      Just r2    -> matchC xs r2
Watching what happens

matchD "" r =
    if nullable r
        then return True
    else error
        ("end of string, but left to matchC: +++show r")
matchD (x:xs) r =
    do { putStrLn (show (x:xs)++" +++show r++"\n")
         ; case consumes1 r x of
             Nothing -> return False
             Just r2 -> matchD xs r2
    }
alpha2 = oneOf "thenls"
ident2 = Concat alpha2 (Star (Union alpha2 digit))

Main> matchD "then12" (Union ident2 key)
"then12"
  ((t+h+e+n+l+s)(t+h+e+n+l+s+0+1+2)*+if+then+else)

"hen12"  ((t+h+e+n+l+s+0+1+2)*+hen)

"en12"   ((t+h+e+n+l+s+0+1+2)*+en)

"n12"    ((t+h+e+n+l+s+0+1+2)*+n)

"12"     ((t+h+e+n+l+s+0+1+2)*+#)

"2"      (t+h+e+n+l+s+0+1+2) *
R.E.’s and NFA’s

• Algorithm that constructs a NFA from a regular expression.

• NFA
  – alphabet, A
  – set of states, S
  – a start state, $S_0$
  – a set of accepting states, $S_F$ subset of S
  – a transition function, $A \times S \rightarrow S$

• Defined by cases over the structure of regular expressions

• Let A, B be R.E.’s, “x” in A, then
  – $\epsilon$ is a R.E.
  – “x” is a R.E.
  – AB is a R.E.
  – A|B is a R.E.
  – A* is a R.E.

1 Rule for each case
Rules

- $\varepsilon$
- "$\chi$"
- $AB$
- $A|B$
Example: \((a|b)^*abb\)

- Note the many \(\varepsilon\) transitions
- Loops caused by the *
- Non-Determinism, many paths out of a state on “a”
Building an NFA from a RE

data Label c = Epsilon | Char c

data Edge alpha state =
       Edge state (Label alpha) state

data NFA alpha state =
       NFA [alpha] -- the alphabet
       [state]   -- set of states
       state     -- start state
       [state]   -- set of final states
       [Edge alpha state] -- transitions
We need to generate new states

- \( nfa :: \text{RE} \rightarrow \text{Int} \rightarrow (\text{Edges}, \text{Int}) \)

- The Int argument and the Int in the result pair, compute the next free state number
nfa :: RE -> Int -> (Edges, Int)
nfa Empty n = ((s, f, [Edge s Epsilon f]), n+2)
  where s = n
       f = n+1
nfa (C x) n = ((s, f, [Edge s (Char x) f]), n+2)
  where s = n
       f = n+1
nfa (Union x y) n = ((s, f, newes ++ xes ++ yes), n2+2)
  where ((sx, fx, xes), n1) = nfa x n
       ((sy, fy, yes), n2) = nfa y n1
       s = n2
       f = n2+1
       newes = [Edge s Epsilon sx, Edge s Epsilon sy
       , Edge fx Epsilon f, Edge fy Epsilon f]
nfa (Concat x y) n = ((sx, fy, edges), n2)
  where ((sx, fx, xes), n1) = nfa x n
          ((sy, fy, yes), n2) = nfa y n1
        edges = (Edge fx Epsilon sy):(xes ++ yes)

nfa (Star r) n = ((s, f, newes ++ res), n1+2)
  where ((sr, fr, res), n1) = nfa r n
        s = n1
        f = n1+1
        newes =
          [Edge s Epsilon sr, Edge fr Epsilon f
           ,Edge s Epsilon f, Edge f Epsilon s]
Constructing an NFA

```haskell
re2nfa re = NFA sigma stateL start [finish] edges
    where ((start,finish,edges),nextState) = nfa re 0
        chars (Edge _ Epsilon _ ) = []
        chars (Edge _ (Char c) _ ) = [c]
        states (Edge s _ f) = [s,f]
        sigma = norm(concat(map chars edges))
        stateL = norm(concat(map states edges))
```
Try it out

Main> example
  (a+b)*abb
Main> re2nfa example

start = 6
final = [13]
[7 -#-> 8
 6 -#-> 4
 5 -#-> 7
 6 -#-> 7
 7 -#-> 6
 4 -#-> 0
 4 -#-> 2
 1 -#-> 5
 3 -#-> 5
 0 -a-> 1
 2 -b-> 3
 9 -#-> 10
 8 -a-> 9
 11 -#-> 12
 10 -b-> 11
 12 -b-> 13
]
Computing over NFA

- Given a set of states, all the states that can be reached from that set on epsilon transitions alone.

- Given a set of states, and a character, all states that can be reached via epsilon transitions and a single transition on the character.
Common function

oneStepByP root p
(NFA sig st start final edges) = norm all
where startsAtRoot (Edge s _ f) = elem s root
possible = filter startsAtRoot edges
meetsP = filter (\ (Edge s c f) -> p c) possible
final (Edge s c f) = f
all = map final meetsP
onEpsilon & onChar

onEpsilon nfa root = norm all
    where p Epsilon = True
          p _ = False
          all = oneStepByP root p nfa ++ root

onChar c nfa root = oneStepByP root p nfa
    where p Epsilon = False
          p (Char x) = c==x
eClosure

- epsilon closure is a fixed point operation.
- We know how to do this.
- Just be sure the set of states is in normal form
- Use the fixpoint operator

```haskell
fixpoint :: Eq a => (a -> a) -> a -> a
fixpoint f init = if init==next
  then next
  else fixpoint f next
  where next = f init

eClosure nfa root = fixpoint (onEpsilon nfa) root

charClosure c nfa root =
  eClosure nfa (onChar c nfa root)
```
recognize (NFA sig st start final edges) stateset []
    = any ( `elem` final) stateset
recognize nfa stateset (x:xs)
    = recognize nfa next xs
    where next = charClosure x nfa stateset

accept (nfa@(NFA sig st start final edges)) string =
    recognize nfa (eClosure nfa [start]) string
\textbf{Main}> \texttt{example}
\texttt{(a+b)*abb}

\textbf{Main}> \texttt{accept (re2nfa example) "abaab"}
\texttt{False}

\textbf{Main}> \texttt{accept (re2nfa example) "abbba"}
\texttt{False}

\textbf{Main}> \texttt{accept (re2nfa example) "ababb"}
\texttt{True}

\textbf{Main}> \texttt{accept (re2nfa example) "abbbabb"}
\texttt{True}
Main> Union ident2 key
((t+h+e+n+l+s)(t+h+e+n+l+s+0+1+2)*+if+then+else)

Main> accept (re2nfa $ Union ident2 key) "then12"
True
Testing

• We now have several (hopefully) equivalent ways of recognizing regular languages.

• We should be able to devise tests that test one form against another.
Overview

- Regular expressions are a language
- Multiple ways to give the language meaning
  - meaning1
  - meaning2
  - maxmunch
  - matchC
  - accept
- The language is a data structure and we can analyze it
  - nullable
  - re2nfa