```
---------------------------------------
-- A module for exploring regular languages in Haskell
-- Fritz Ruehr • Willamette CS 465 • Spring 2015
---------------------------------------

module RegLang where

import Data.List ((\\))  -- list difference

---------------------------------------
-- A data type for regular expressions (REs)

data RE = Null | Epsi | Lit Char | Bar RE RE | Dot RE RE | Star RE

---------------------------------------
-- Printing REs (with full parens, and just ASCII)

instance Show RE where
  show  Null     = "0"
  show  Epsi     = "e"
  show (Lit a)   = [a]
  show (Bar p q) = pop '|' (show p) (show q)
  show (Dot p q) = pop '.' (show p) (show q)
  show (Star p)  = par (show p ++ "*")

pop c x y = par (x++[c]++y)
par s = "(" ++ s ++ ")"

---------------------------------------
-- Matching: a predicate semantics for REs based on string partition

match  Null     s = False
match  Epsi     s = null s
match (Lit a)   s = s==[a]
match (Bar p q) s = match p s || match q s
match (Dot p q) s =           any (\(x,y) -> match p x && match q y)        (part s)
match (Star p)  s = null s || any (\(x,y) -> match p x && match (Star p) y) (part s)

part [] = [([],[])]
part (x:xs) = ([],(x:xs)) : map (lft (x:)) (part xs)

---------------------------------------
-- Sample REs (incl. sheep, Swedish rock, and Beach Boys)

a  = Lit 'a'        -- a
b  = Lit 'b'        -- b
ab = Bar a b        -- (a|b)

plus r = Dot r (Star r)

axa = a `Dot` (Star ab) `Dot` a                -- a.(a|b)*.a
axb = a `Dot` (Star ab) `Dot` b                -- a.(a|b)*.b
baa = Star (b `Dot` a `Dot` plus a)            -- (b.a.a+)*
swd = plus (a `Dot` b `Dot` plus b `Dot` a)    -- (a.b.b+.a)+
bch = plus (b `Dot` a)                         -- (b.a)+

---------------------------------------
-- A data type for deterministic finite automata (DFAs)

data (Eq q, Eq s) => DFA q s = DFA [q] [s] (q->s->q) q [q]
```

```
--------------------------------------
-- Validity check and acceptance function

check (DFA qs s d q f) = all (`elem` qs) (q:f ++ [d q a | q<-qs, a<-s])

acc m@(DFA qs s d q f) str = if check m then elem (foldl d q str) f
                                        else error "invalid DFA"

--------------------------------------
-- DFA constructions: negation and product (rel. to a boolean operator)

neg (DFA qs s d q f) = DFA qs s d q (qs \\ f)

prod op (DFA qs s d q f) (DFA rs t e r g) =
  if s /= t then error "alphabetic mis-match"
            else (DFA cross s d' (q,r) (comb op))

  where cross  = [ (q,r) | q<-qs, r<-rs ]
        comb p = [ (q,r) | q<-qs, r<-rs, p (elem q f) (elem r g) ]
        d' (q,r) s = (d q s, e r s)

--------------------------------------
-- Sample DFAs (corresponding to REs baa and bch above)

baam = DFA [1..5] "ab" d 1 [1,4]
       where d 1 'a' = 5 ;   d 1 'b' = 2
             d 2 'a' = 3 ;   d 2 'b' = 5
             d 3 'a' = 4 ;   d 3 'b' = 5
             d 4 'a' = 4 ;   d 4 'b' = 2
             d 5 'a' = 5 ;   d 5 'b' = 5

bchm = DFA [1..4] "ab" d 1 [3]
       where d 1 'a' = 4 ;   d 1 'b' = 2
             d 2 'a' = 3 ;   d 2 'b' = 4
             d 3 'a' = 4 ;   d 3 'b' = 2
             d 4 'a' = 4 ;   d 4 'b' = 4

--------------------------------------
-- Utility functions: cut a list by a predicate, "winning" strings

cut p [] = ([],[])
cut p (x:xs) = (if p x then lft else rgt) (x:) (cut p xs)

win p = fst . cut p

lft f (x,y) = (f x,y)
rgt f (x,y) = (x,f y)

--------------------------------------
-- Generating strings over {a,b} in various ways, for testing

gen 0 = [[]]
gen k = map ('a':) g ++ map ('b':) g where g = gen (k-1)

abs k =         concatMap gen [0..k]
get k = take k (concatMap gen [0..])

--------------------------------------
-- End of module RegLang
--------------------------------------
```