Lesson 1

```
{-   The Finonacci numbers
     Note: this sequence was known by the Indian mathematician Pingala in 350bc!
          Fibonacci simply rediscovered them ...
     [from wikipedia:]
     Fibonacci considers the growth of a hypothetical, idealized (biologically unrealistic) rabbit population,
     assuming that: a newly born pair of rabbits, one male, one female, are put in a field; rabbits are able
     to mate at the age of one month so that at the end of its second month a female can produce another pair
     of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female)
     every month from the second month on. Fibonacci posed the puzzle: how many pairs will there be in
     one year?

     Answer: 144 -}

--  (0) Basic Fibonaccci:

--fib:: Integer -> Integer  -- (don't forget to specify the type!!!)
fib 0 = 0
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))

-- What happens with (a) fib 50 (b) fib (-1) (c) fib 2.5????

--  (1) Basic Fibonacci with guards and simple error handling

fib1:: Integer -> Integer
fib1 n
   | n < 0     = error "Positive numbers only please ..."
   | n == 0    = 0
   | n == 1    = 1
   | otherwise = (fib1 (n-1)) + (fib1 (n-2))

{-  Here is a trace of fib1 3
    [fib1 3]
    3 < 0  fail
    3 == 0 fail
    3 == 1 fail
    [fib1(3-1) + fib1(3-2)]
    [fib1(3-1)] + fib1(3-2)
     fib1 [3-1]
      2
     [fib1 2]
     2 < 0 fail
     2 == 0 fail
     2 == 1 fail
```

```
    otherwise succeed
    [fib1(2-1) + fib1(2-2)]
    [fib1(2-1)] + fib1(2-2)
     fib1 [2-1]
      1
     [fib1 1]
     1 < 0 fail
     1 == 0 fail
     1 == 1 succeed
     [1]
    1 + [fib1(2-2)]
     [2-2]
      0
     [fib1 0]
     0 == 0 succeed
      [0]
     [1 + 0]
     1
    1 + [fib1(3-2)]
     [3-2]
    [1 + 1]
    2 -}

-- (3) Fast Fibonacci:
-- This is a more efficient version which does not repeat calculations
-- This uses a where clause

fibstep (n,m) = (m,n+m)
dec n = n-1

ffib:: Integer -> Integer
ffib n
  | n < 0 = error "Fibonacci only takes positive integers"
  | otherwise = pfib n (0,1)
 where
  pfib 0 (n,m) = n
  pfib r (n,m) = pfib (r-1) (fibstep (n,m))

-- (4) Or you can write this with a let clause
ffib':: Integer -> Integer
ffib' n
  | n < 0 = error "Fibonacci only takes positive integers"
  | otherwise =
     let
       pfib 0 (n,m) = n
       pfib r (n,m) = pfib (r-1) (fibstep (n,m))
```

```
    in pfib n (0,1)


--(5) Or you can use an anonymous function
fibP 0 = (0,1)
fibP n = (\(n,m) -> (m,n+m)) (fibP (n-1))

-- (6) )r you could use the composition operator `.'
fibP' 0 = (0,1)
fibP' n = (fibstep . fibP'. dec) n


-- (7) The more efficient version which does not repeat calculations
-- Can be written with composition and a where clause in this form
ffibc:: Integer -> Integer
ffibc n
  | n < 0 = error "Fibonacci only takes positive integers"
  | otherwise = pfib n (0,1)
 where
  pfib 0 = fst
  pfib r = pfib (r-1) . fibstep


-- (8) The average of three numbers (for an example of currying)
average::Float -> Float -> Float -> Float
average a b c = (a+b+c)/3

-- (9) doubling an integer
double:: Integer -> Integer
double n = n + n

-- (10) iterating a function
myiterate:: Integer -> (a -> a) ->  a -> a
myiterate 0 f = id
myiterate n f = myiterate (n-1) f . f

-- (11) Exponential
exponential:: Integer -> Integer
exponential n = myiterate n double 1

-- (11) Fibonacci again using iteration!
fibonacci:: Integer -> Integer
fibonacci n = (fst. myiterate n fibstep) (0,1)

Lesson 2
{-
```

```
   Scott's bottom: this is an "element" of each type which never terminates.  It is,
   thus, an "undefined" or "divergent" element.  Any program which causes bottom to be
   evaluated will also not terminate ... SO one can test whether a program "touches" an
   argument with this function!!

   It is named after the great computer scientist, philosopher, and mathematician Dana Scott.
-}

bottom::a
bottom = bottom

--  data Bool = True | False
--     deriving (Eq,Show)

-- different versions of (&&) !
-- First this is how it is programmed in the prelude
--     ... we use the "section" of the infix operator
--     so we can program it just like any other function
--
-- (&&):: Bool -> Bool -> Bool
-- (&&) True True = True
-- (&&) _     _   = False

-- different versions of and which have different behaviors

myand1:: Bool -> Bool -> Bool
myand1 True True = True
myand1 True False = False
myand1 False True = False
myand1 False False = False

myand2:: Bool -> Bool -> Bool
myand2 _ False = False
myand2 False _ = False
myand2 True True = True

-- Try False && bottom ... why do the other versions of  and
-- behave slightly differently???

{- In the prelude there is the datatype
     data Maybe a = Nothing | Just a
                deriving (Eq,Ord,Read,Show)
   here we create our own "Maybe" datatype called
   "success or fail" written "SF" and play with it.  -}

data SF a = SS a | FF
```

```
data SF a = SS a | FF
    deriving (Eq,Show)

-- This allows us a cleaner way to catch the error when one enters a negative
-- in Fibonacci:

fibstep (n,m) = (m,n+m)

myiterate:: Integer -> (a -> a) -> a -> a
myiterate 0 f = id
myiterate n f = myiterate (n-1) f . f

fibsf:: Integer -> (SF Integer)
fibsf n | n < 0 = FF
        | n < 2 = SS 1
        | otherwise = SS ((fst.myiterate n fibstep) (0,1))

-- mapping the success or fail data to booleans
sf_2_bool:: SF a -> Bool
sf_2_bool (SS x) = True
sf_2_bool FF = False

-- try sf_2_bool (SS bottom) !!!

--  division with exceptions (note the explicit coercion
--  of integers into floating point)

sfdivide:: Int -> Int -> (SF Float)
sfdivide n m | m == 0 = FF
             | otherwise = SS ((fromIntegral n)/(fromIntegral m))

--  Composing Maybe/Success or Fail functions ---

(>/>):: (a -> (SF b)) -> (b -> (SF c)) -> (a -> (SF c))
(>/>) f g = \x -> case f x of
                    FF -> FF
                    SS y -> g y

-- map for Maybe/Success or Fail
sfmap:: (a -> b) -> ((SF a) -> (SF b))
sfmap f FF = FF
sfmap f (SS x) = SS (f x)



--
-- Initial foray into lists
```

```
--

myhead::[a] -> (SF a)
myhead [] = FF
myhead (a:_) = SS a

mytail::[a] -> (SF [a])
mytail [] = FF
mytail (x:xs) = SS xs

myappend:: [a] -> [a] ->[a]
myappend [] ys = ys
myappend (x:xs) ys =x:(myappend xs ys)

flatten:: [[a]] -> [a]
flatten [] = []
flatten (xs:xss) = myappend xs (flatten xss)

mymap::(a  -> b) -> ([a] -> [b])
mymap f [] = []
mymap f (x:xs) = (f x):(mymap f xs)

mylength [] = 0
mylength (x:xs) = 1 + (mylength xs)

-- Try [bottom,bottom,bottom]
-- Try length [bottom,bottom,bottom] ...
-- Try myappend [bottom,bottom] [bottom,bottom,bottom]
-- Try length (myappend [bottom,bottom] [bottom,bottom,bottom])

Lesson 3
{- We LOVE lists! (part 1)

  Lists are probably the most commonly used datatype of any language ..
  For this reason in Haskell there is a special builtin syntax for lists.

   [] is the empty list
   x:xs is the list with first element x and "tail" the list xs
   [1,2,3,4,5,6] an example of a list of integers
   [a] is (besides the one elemnet list) the type of list of a's

-}

-- appending lists
myappend:: [a] -> [a] -> [a]
myappend [] ys = ys
```

```
myappend (x:xs) ys = x:(myappend xs ys)

-- in the prelude append is defined as an operator
--   (++) :: [a] -> [a] -> [a]
--   [] ++ ys = ys
--   (x:xs) ++ ys = x:(xs ++ ys)

myflatten::[[a]] -> [a]
myflatten [] = []
myflatten (xs:xss) = myappend xs (myflatten xss)

-- in the prelude this is concat BUT look up its type!!

mylength:: [a] -> Integer
mylength [] = 0
mylength (x:xs) = 1 + (mylength xs)

-- in the prelude this is length

mymember:: Eq a => a -> [a] -> Bool
mymember a [] = False
mymember a (x:xs) | a==x = True
                  | otherwise = mymember a  xs

--mapping a function over a list
mymap:: (a -> b) -> ([a] -> [b])
mymap f [] = []
mymap f (a:as) = (f a):(mymap f as)

--given a list of booleans compute their conjunction
myAND:: [Bool] -> Bool
myAND [] = True
myAND (b:bs) = b && (myAND bs)

--given a list of booleans compute their disjunction
myOR:: [Bool] -> Bool
myOR [] = False
myOR (b:bs) = b || (myOR bs)

-- Checking whether there  is something in the list which satisfies
-- a predicate
inlist:: (a -> Bool) -> [a] -> Bool
inlist pred as = myOR (mymap pred as)

Lesson4
{- We LOVE lists (part 2)
```

```
    You can do alot with lists!  But they are by no means the end of the story:
    they are just one well-known datatype ...

    All datatypes come with a "map" function and a "fold" function.
    The fold function (often called "fold left" for lists) is particularly
    fundamental ...

    These notes begin to explore this aspect of lists ... and end by describing
    list comprehension -- a powerful feature which has leaked into other
    languages

-}

data SF a = SS a | FF
  deriving (Eq,Show)

sfmap:: (a -> b) -> ((SF a) -> (SF b))
sfmap f (SS x) = SS (f x)
sfmap f FF = FF

--mapping a function over a list
mymap:: (a -> b) -> ([a] -> [b])
mymap f [] = []
mymap f (a:as) = (f a):(mymap f as)

--given a list of booleans compute their conjunction
myAND:: [Bool] -> Bool
myAND [] = True
myAND (b:bs) = b && (myAND bs)

--given a list of booleans compute their disjunction
myOR:: [Bool] -> Bool
myOR [] = False
myOR (b:bs) = b || (myOR bs)

-- Checking whether there  is something in the list which satisfies
-- a predicate
inlist:: (a -> Bool) -> [a] -> Bool
inlist pred as = myOR (mymap pred as)

-- mymember again ..
mymember:: Eq a => a  -> [a] -> Bool
mymember a = inlist (\x -> a==x)

-- Is there a number greater than m in the list?
```

```
myGT:: Integer -> [Integer] -> Bool
myGT m = inlist (\x -> x>m)

-- Zipping two lists: what happens if they are different lengths?
-- We want the zip to fail!!

myzip:: [a] -> [b] -> SF [(a,b)]
myzip [] [] = SS []
myzip (a:as) (b:bs) =
     case myzip as bs of
       SS xs -> SS ((a,b):xs)
       FF -> FF
myzip _ _ = FF

-- try myzip [1,2,3] "abc"!!  Point is "abc = ['a','b','c']

-- Zipping with two lists and applying a function f

myzipwith:: ((a,b) -> c) -> [a] -> [b] -> (SF [c])
myzipwith f as bs = sfmap (mymap f) (myzip as bs)

-- Currying and uncurrying -- and who was Haskell Curry anyway?
-- we can typs a function in two subtely different ways:
--       f:: a -> b -> c   or   f':: (a,b) -> c
--the first is a "curried" version the second is "uncurried"
-- they are interdefinable
--       f a b = f'(a,b)   and   f'(a,b) = f(a,b)
-- however what we cannot (immediately) do with the second function
-- is to get a function
--            (f a) :: b -> c
-- to do this for f' we have to explicitly "curry" it
--            g a = (\b -> f'(a,b)):: b -> c

-- An important and useful function is "filtering" a list:
-- this removes elements from a list which do not satisfy a
-- predicate ...

myfilter:: (a -> Bool) -> [a] -> [a]
myfilter p [] = []
myfilter p (a:as) = if (p a) then a:(myfilter p as)
                    else myfilter p as

-- Note "if then else" could have been replaced by guards
--        myfilter p (a:as)| p a = a:(myfilter f as)
--                         | otherwise = myfilter f as
-- much nicer!!!
```

```
-- Reversing a list:
-- The naive reverse: complexity O(n^2) ... ugh!

n_reverse:: [a] -> [a]
n_reverse [] = []
n_reverse (a:as) = (n_reverse as)++[a]

-- Reversing a list the "fast" way: complexity O(n) .. better!

f_reverse:: [a] -> [a]
f_reverse xs = shunt xs []
  where
    shunt:: [a] -> [a] -> [a]
    shunt [] ys = ys
    shunt (x:xs) ys = shunt xs (x:ys)

-------------------------------------------------
-- folding on a list (foldl in prelude)
-- here is a more sophisticated but - if you can get
-- your mind round it - very useful higher order
-- function.
-------------------------------------------------

myfold:: (a -> c -> c) -> c -> ([a] -> c)
myfold f c [] = c
myfold f c (x:xs) = f x (myfold f c  xs)


--  adding up a list of integers
myADD:: [Integer] -> Integer
myADD = myfold (+) 0

-- multiplying a list of  integers
myMUL:: [Integer] -> Integer
myMUL = myfold (*) 1

-- appending two list
myAPP:: [a] -> [a] -> [a]
myAPP as bs = myfold (:) bs as

-- flattening a list
myFLATTEN:: [[a]] -> [a]
myFLATTEN = myfold myAPP []

-- another version of map for a list
```

```
myfmap:: (a -> b) -> ([a] -> [b])
myfmap f = myfold (\a as -> (f a):as) []

myffilter:: (a -> Bool) -> [a] -> [a]
myffilter p = myfold
              (\a as -> if (p a) then a:as else as)
              []

-- This is a higher-order reverse for you to puzzle over
-- (it will not be on any tests)

myfreverse :: [a] -> [a]
myfreverse as = myfold (\ a f -> (\x -> f(a:x))) id as []



-------------------------------------------
--  List comprehension is powerful
--   ... but can lead to not very
--  efficient code!
-------------------------------------------


pairs:: [a] -> [b] -> [(a,b)]
pairs as bs = [(a,b) | a <- as, b <- bs]

-- the list comprehension of the above translates out to
-- give (roughly) the following code:

pairs1:: [a] -> [b] -> [(a,b)]
pairs1 as bs = myFLATTEN (mymap (\a -> mymap (\b -> (a,b)) bs) as)

-- the filter function can be written using list comprehension

myfilter1:: (a -> Bool) -> [a] -> [a]
myfilter1 f as = [ a | a <- as, f a]

-- This list coprehension translates out to

myfilter3:: (a -> Bool) -> [a] -> [a]
myfilter3 p as = myFLATTEN (mymap (\a -> if p a then [a] else []) as)

--
-- Here is the translation scheme for list comprehension
--      {[s | a <- t, rest]} = flatten (map (\a -> {[ s | rest]}) t)
```

```
--      {[s | p, rest]} = if p then {[s | rest]} else []
--      {[s | ]} = [s]  (rest is empty!)
--
{-
  Two example of translation:
(1) RHS of pair:
  {[ (a,b) | a <- as, b <- bs ]}
        = flatten (map (\a -> {[ (a,b)| b <- bs ]}) as )
        = flatten (map (\a -> flatten( map (\b -> {[ (a,b)| ]}) bs)) as)
        = flatten (map (\a -> flatten( map (\b -> [(a,b)]) bs)) as)
(2) of RHS of filter:
  {[ a | a <- as, f a]}
        = flatten (map (\a -> {[a| f a]}) as)
        = flatten (map (\a -> if (f a) then {[ a| ]} else []) as)
        = flatten (map (\a -> if (f a) then [a] else []) as)
-}




------------------------------------------------------------
-- Here are four classic examples of using list comprehension
--
-- Quick sort using list comprehension
-- Warning this is not the most efficient quick sort!!
--        ... however, it is pretty elegant!
--

qsort:: (Ord a) => [a] -> [a]
qsort [] = []
qsort (a:as) = (qsort [y|y <- as, y < a])++[a]++(qsort [y|y <- as, y >= a])

--
-- Finding all Pythagorean triples that is (x,y,z) such that x^2 + y^2 = z^2
-- with z =< n
--         ... not very efficient (but elegant)!
--

pythag :: Integer -> [(Integer,Integer,Integer)]
pythag n = [(x,y,z) | z <- [1..n], y <- [1..z], x <- [1..y], x*x+y*y==z*z]

--
-- Finding all prime numbers
-- This implements the sieve of Eratosthenes
--   ( https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)
                very inefficient but elegant
```

```
--          .. very inefficient but elegant
--


primes:: [Integer]
primes = filterPrime [2..] where
   filterPrime (p:xs) = p:(filterPrime [x|x <- xs, x `mod` p /= 0])


--
--  A Mersenne prime is a prime number that is one less than a power of two.
--  Find all the Mersenne primes ...
--        https://www.mersenne.org/primes/
--  How many can you discover?  May want to speed this up ...
--    (it is not even known whether there are infinitely many!!)
--


mersenne:: [Integer]
mersenne = [ n | n <- primes, mestest n ]  where
   mestest n = powertwo (n+1)
   powertwo n    | n == 1 = True
                 | n `mod` 2 == 0 = powertwo (n `div` 2)
                 | otherwise = False


--
-- Just crazy!!  Try running this for a week!


Lesson 5
{-  Probably the most significat feature of functional
   programming languages is their datatypes.  Lists are
   all very fine but we must get past them ... :-)

    So now we dip into data types ..

    The simplest datatype is Bool

     data Bool = True | False

    But what next ... here we dip into non-empty list:
    they are  nearly lsits but not quite!!
-}



--
--  Without success/fail nothing happens!
--
data SF a = SS a
          | FF
```

```
  deriving (Show, Eq)

-- Non-empty lists  ... they are almost list but not quite

data NList a = Single a
             | Many a (NList a)


instance Show a => Show (NList a) where
         show xs = "[["++(showrest xs) where
              showrest (Single a) = (show a)++"]]"
              showrest (Many a ys) = (show a)++","++(showrest ys)

instance Eq a => Eq (NList a) where
          (Single a) == (Single b) = a==b
          (Many a as) == (Many b bs) = (a==b) && (as == bs)

-- Non empty lists are nearly lists
--     in fact non-empty lists wrapped in the "success or fail"
--     datatype are isomorphic to lists!!
--     Here are the translations in each direction ...
nl2l:: SF (NList a) -> [a]
nl2l FF = []
nl2l (SS xs) = nl2l_helper xs where
     nl2l_helper (Single x) = [x]
     nl2l_helper (Many x xs) = x:(nl2l_helper xs)

l2nl:: [a] -> (SF (NList a))
l2nl [] = FF
l2nl (x:xs) = SS (l2nl_helper x xs) where
      l2nl_helper x [] = Single x
      l2nl_helper x (y:ys) = Many x (l2nl_helper y ys)

-- Here is the "fold" for non-empty lists

foldNList:: (a -> b -> b) -> (a -> b) -> (NList a) -> b
foldNList f g (Single x) = g x
foldNList f g (Many x xs) = f x (foldNList f g xs)

-- Now we can sum non-empty lists easily

sumNList:: (NList Integer) -> Integer
sumNList = foldNList (+) id

-- We can append non-empty lists
appNList:: (NList a) -> (NList a) -> (NList a)
```

```
appNList as bs = foldNList Many (\a -> Many a bs)  as

{- For testing!!

test_appNList as bs =
  case (l2nl as,l2nl bs) of
    (SS xs,SS ys) -> SS (appNList xs ys)
    _  -> FF

test_appNList "abcd" "efgh"
-}

-- We can flatten non-empty lists
flattenNList:: (NList (NList a)) -> NList a
flattenNList = foldNList appNList id

--
-- one thing we can do for non-empty lists which we
-- cannot REALLY do for lists is to take the head!
--

headNList:: (NList a) -> a
headNList (Single a) = a
headNList (Many a _) = a



-----------------------------------------------------
--
--  Lets talk natural numbers: this is a really
--        foundational datatype -- the "unary numbers".
--        It is not very practical ... but conceptually
--        crucial!
--
-----------------------------------------------------

data Nat = Succ Nat | Zero
   deriving (Eq)

instance Show Nat where
      show Zero = "z"
      show (Succ n) = "s"++(show n)
--
--  First up what is a fold for Nat?
--
foldNat:: (a -> a) -> a -> Nat -> a
```

```
foldNat f a0 Zero = a0
foldNat f a0 (Succ n) = f (foldNat f a0 n)

--  Note this is a "for loop"  n |-> f^n(a0) ... done in
--  unary!

-- Transalating from integers and back

nat2int:: Nat -> Integer
nat2int = foldNat (\n -> n+1) 0

int2nat::Integer -> (SF Nat)
int2nat n | n < 0 = FF
          | otherwise = SS (int2nat' n) where
    int2nat' 0 = Zero
    int2nat' n = Succ (int2nat' (n-1))

-- Adding and multiplying in unary

addNat:: Nat -> Nat -> Nat
addNat n m = foldNat Succ m n

multNat:: Nat -> Nat -> Nat
multNat n = foldNat (addNat n) Zero

-- the predecessor function and monus

mypred:: Nat -> Nat
mypred Zero = Zero
mypred (Succ m) = m

monus:: Nat -> Nat -> Nat
monus n = foldNat mypred n

------------------------------------------------------
--
--  Lets talk binary numbers (this is essentially what
--  is implemented in hardware: so these numbers are
--  seriously built-in!!
--
------------------------------------------------------

data BNat = B0 BNat | B1 BNat | BZ
   deriving (Eq)

instance Show BNat where
```

```
    show m = "{"++(show' m)++"}" where
        show' BZ = ""
        show' (B0 m) = "0"++(show' m)
        show' (B1 m) = "1"++(show' m)
--
-- Again lets get the fold sorted:
--
foldBNat:: (a -> a) -> (a -> a) -> a -> BNat -> a
foldBNat f0 f1 a BZ = a
foldBNat f0 f1 a (B0 n) = f0 (foldBNat f0 f1 a n)
foldBNat f0 f1 a (B1 n) = f1 (foldBNat f0 f1 a n)


--
-- Translating to and from Integers:
--

bnat2int:: BNat -> Integer
bnat2int = fst . (foldBNat (\(n,b) -> (n,2*b)) (\(n,b) -> (n+b,2*b)) (2,0))

int2BNat:: Integer -> (SF BNat)
int2BNat n | n<0 = FF
           | otherwise = SS (int2BNat' n BZ) where
  int2BNat' n | n==0 = id
              | (n `mod` 2) == 1 = (\x -> (int2BNat' (n `div` 2)) (B1 x))
              | (n `mod` 2) == 0 = (\x -> (int2BNat' (n `div` 2)) (B0 x))
              | otherwise = id


--
--  Can you  write addition and multiplication?
--


Lesson6
--
--   WE DON'T LOVE TREES YET [Part I]
--
--           ... and why all this folding?
--


data Tree a = Branch (Tree a) (Tree a)
            | Leaf a
  deriving (Show,Eq)


--
--  Example elements of the type Tree a are
--         Leaf a
--         Branch (Leaf a1) (Leaf a2)
```

```
--         Branch (Leaf a1) (Leaf a2))
--         Branch (Branch (Leaf a1) (Leaf a2)) (Leaf a3)
--         ...
--


--
-- Here is the fold for Tree with a function for the Branch
-- constructor, f_b, and a function, f_l, for the Leaf
-- constructor:
--
foldTree:: (c -> c -> c) -> (a -> c) -> (Tree a) -> c
foldTree f_b f_l (Leaf a) = f_l a
foldTree f_b f_l (Branch t1 t2)
    = f_b (foldTree f_b f_l t1) (foldTree f_b f_l t2)


--
-- To sum the leaves of a Tree of Integers
--
sumTree:: Tree Integer -> Integer
sumTree = foldTree (+) id

-- A common thing to want to do is to traverse the leaves of a tree.
-- A step to facilite this is to collect the leaves into a list:

collect:: Tree a -> [a]
collect = foldTree (++) (\x -> [x])


-------------------------------------------------------------------------
--              This brings up some COMPLEXITY THOUGHTS
--
--  The complexity of this collect is determined by the complexity of
--  (++) which, in turn, is determined by the size of its first argument.
--  Thus consider a tree of uniform depth d (which will have 2^d leaves and
--  2^d-1 branch nodes) then a node at depth r will have to append two lists
--  of length 2^(d-r) however there are 2^r of these nodes. Thus the work
--  done at each level is 2^(d-r) * 2^r = 2^d/2 and there are d-1 level
--  of nodes.  Thus the cost of the computation is d * 2^d.
--
--  Finally setting the size of the tree, n, to be the number leaves
--  then cost of the computation is n * (log n)  which is not bad.
--
--  However if the tree is skewed to the left (so all right branches
--  are leaves) then the cost of the computation becomes
--          1+2+..+(n-1) = n*(n-1)/2
--  which is quadratic and not so good :-/
--
    So the downside of this algorithm is that in the worst case it is
```

```
-- So the downside of this algorithm is that in the worst case it is
-- quadratic.
--
-- Can we do anything about this? In fact with a higher-order trick
-- we can reduce the complexity to O(n):

fcollect:: Tree a -> [a]
fcollect t = foldTree (.) (\a x -> a:x) t []

-- Here the operation at each noode of the tree is composition of functions
-- which is a constant time operation (so complexity O(n)). The last step
-- is to evaluate this sequence of compositions of functions on the
-- empty list [] (complexity O(n)). So the whole has O(n) complexity!

-- (NB. Haskell program optimization can automatically catch this sort
-- of in efficency.)
--------------------------------------------------------------------------

--
-- The success or fail datatype (again)
data SF a = SS a| FF
   deriving (Show,Eq)

mapsf:: (a->b) -> (SF a) -> (SF b)
mapsf f (SS a) = SS (f a)
mapsf _ FF = FF

sflist:: [SF a] -> (SF [a])
sflist = foldr (\a as -> case (a,as) of
                         (SS x, SS xs) -> SS (x:xs)
                         _ -> FF)
           (SS [])

-----------------------------------------
--
-- The datatype of expressions:
--
-----------------------------------------

data Exp f v  = Opn f [Exp f v]
           | Var v
   deriving (Show,Eq)

-- Typical elements of (Exp String Integer) are
--       Var 3
--       Opn "add" [Var 2,Var 3,Var 4]
--       Opn "div" [Var 7, Opn "add" [Var 9,Var 7]]
```

```
--        Opn "div" [Var 7, Opn "add" [Var 9,Var 7]]
--        Opn "p" [Opn "q" [],Opn "r" [Var 6, Opn "s" []]]
--        ..
-- We think of this as an abstract "expression" with opperation names
-- "p", "q", "r" acting on arguments

--
-- Here is the fold for the datatype (note the use of the map
-- to get inside the arguments!
--
foldExp:: (f -> [c] -> c) -> (v -> c) -> (Exp f v) -> c
foldExp opn var (Var v) = var v
foldExp opn var (Opn f args) = opn f (map (foldExp opn var) args)

--
-- To illustrate how we may view this as an expression
-- suppose we  have operations of addition and multiplication
-- Then we can "evaluate" an expression of integers

data Operations = Add | Mult
   deriving (Show,Eq)

evaluate:: Exp Operations Integer -> Integer
evaluate = foldExp (\x -> case x of
                          Add -> foldl (+) 0
                          Mult -> foldl (*) 1)
             id

-------------------------------------------------
--
--            MATCHING
--
--    We have already met the idea of matching in hand evaluating
-- Haskell programs.  However, we can actually program it!!
-- The idea is given two expression trees the second of which
-- we view as a template we can find the substitutions of the
-- variables in the second tree in order to reconstruct the
-- first tree.  Of course, matching may fail ...  so we shall
-- return a success or fail of a list of substitutions.
--
-- Here is the basic matching program:

match:: (Eq f) => (Exp f v) -> (Exp f w) -> (SF [(w,Exp f v)])
match t (Var v) = SS [(v,t)]
match (Opn f1 args1) (Opn f2 args2)
  | f1==f2 && (length args1 == length args2)
```

```
        = mapsf concat
          (sflist (map (\(t1,t2) -> match t1 t2) (zip args1 args2)))
  | otherwise = FF

--  Try this out on
--     match (Opn "f" [Opn "g" [],Opn "h" [Var 3,Var 4],Var 5])
--           (Opn "f" [Var "x",Var"y"])
--  then try
--     match (Opn "f" [Opn "g" [],Opn "h" [Var 3,Var 4],Var 5])
--           (Opn "f" [Var "x",Var"y",Var "z"])
--  What happened?
--
-- Now there is a another wrinkle to pattern matching
-- (as opposed to match ching) which is that the
-- template must not have repeated variables ...
-- we really should check for this!!
-- We can do so in two steps
--   (1) Obtain a list of variable
--   (2) check that the list is not repeating
-- Now (1) can be achieved very simply using a fold:

collectvar:: Exp f v -> [v]
collectvar = foldExp (\f vargs -> concat vargs) (\x -> [x])

-- For (2) we have

nonrepeating:: Eq a => [a] -> Bool
nonrepeating (a:as) = (nonrepeating as) && not (member a as)
   where member a [] = False
         member a (b:bs)| a==b = True
                        | otherwise = member a bs
nonrepeating [] = True

-- Note that this test is quadratic: every (unordered) pair of
-- elements of the list must be compared.

-- So now we can put this together into a "pattern matching"

pmatch:: (Eq f,Eq v,Eq w) => (Exp f v) -> (Exp f w) -> (SF [(w,Exp f v)])
pmatch t template| nonrepeating (collectvar template) = match t template
                 | otherwise = FF

-- This time pattern matching also fails if one does not have a "legal"
-- template!
--
--  Try this out on
```

```
--      match (Opn "f" [Opn "g" [],Opn "h" [Var 3,Var 4],Var 5])
--            (Opn "f" [Var "x",Var "x"])
```

Lesson7

OK we don't love trees yet ...

... BUT we can see they MIGHT be useful!!

Please note that I have flipped to using "literate Haskell" this because I am writing more comments than program!!

OK so we need to start thinking about game playing!!!!

The ideas behind game playing games like checkers where pioneered by Arthur Samuel. His work actually focused particularly on evaluating game states (often called the "heuristic"). The basic idea is that one chooses the move that maximizes the heuristic value of the game state reached by each possible move.

Now to increase the efficacy of these heuristics one combines them with looking ahead at all the possible plays in the game upto some depth. A single move (of the opponent or player) in the game is called a "ply": thus one looks ahead say 8 ply (this is quite a lot!!).

When one looks ahead in this manner one sweeps out a "MinMax" tree. The tree branches first correspond to the players possible moves, then correspond to the opponents moves etc. It is called a minmax tree because the opponent tries to minimize the (backed up) heuristic while the player tries to maximize the (backed up) heuristic.

Here is the datatype of a minmax tree in Haskell:

```
> data MinMax a = TipMin Int a
>               | BranchMin [MaxMin a]
>       deriving (Show ,Eq)

> data MaxMin a = TipMax Int a
>               | BranchMax [MinMax a]
>       deriving (Show,Eq)
```

These MinMax trees are mutually recursive datatypes: "MinMax a" calls "MaxMin a" and "Maxmin a" calls "MinMax a".

When searching a real game tree a tree such as the above is never built. Instead the tree is implicit in the calculation. However, the datatype above is useful in developing the idea of how

one eveluates a game tree. The heuristic is evaluated at the tips of the tree (usually at some fixed ply). The player wants to maximize the reward/heuristic at their turn while the opponent, at their turn, tries to minimize the reward/heuristic that the player can achieve.

Thus fundamental to game playing is the evaluation of these MinMax trees:

```
> maxeval:: MaxMin Integer -> Integer
> maxeval (TipMax _ n) = n
> maxeval (BranchMax ms) = foldl max bot (map mineval ms)

> mineval:: MinMax Integer -> Integer
> mineval (TipMin _ n) = n
> mineval (BranchMin ms) = foldl min top (map maxeval ms)
```

Here is a test MinMax tree to try out the algorithm:

```
> test = BranchMax
>          [ BranchMin [TipMax 1 7,BranchMax [TipMin 2 9,TipMin 3 2]]
>          , BranchMin [BranchMax [TipMin 4 18,TipMin 5 9], TipMax 6 31]
>          , BranchMin [TipMax 7 17,
>                       BranchMax [BranchMin [TipMax 8 12,TipMax 9 17],
>                                  TipMin 10 45],
>                       TipMax 11 72]
>          ]
```

Very early in the development in game playing it was realized that the evaluation of MinMax trees could be optimized. The optimization ensures that one does not evaluate parts of the tree which will make no difference to the result.

The idea is to bound the answer a < ans < b and adjust these bounds as one does the evaluation. One starts the process with some "global" bounds bot < ans < top.

When one is maximizing one can increase the lower bound, while one is minimizing one can decrease the upper bound. This gives the following basic \alpha-\beta pruning algorithm:

```
> top = 30000
> bot = -30000

> abmaxprune:: (MaxMin Integer) -> Integer -> Integer -> Integer
> abmaxprune t a b| a==b = a -- cutoff
>                | otherwise = case t of
>                    (TipMax i v) -> min (max v a) b
```

```
>                   (BranchMax ts) -> abmaxprunes ts a b
> abmaxprunes [] a b = a
> abmaxprunes (t:ts) a b = abmaxprunes ts newa b
>     where
>       newa = abminprune t a b

> abminprune::(MinMax Integer) -> Integer -> Integer -> Integer
> abminprune t a b| a==b = b --cutoff
>                 | otherwise = case t of
>                   (TipMin i v) -> max (min v b) a
>                   (BranchMin ts) -> abminprunes ts a b

> abminprunes [] a b = b
> abminprunes (t:ts) a b = abminprunes ts a newb
>     where
>       newb = abmaxprune t a b
```

In order to see the effect of \alpha-\beta pruning we shall "instrument"
the above code to show which "tips" have been evaluated. This was why
we numbered the nodes of the tree. In the above example note that
there are 11 "tips" but with pruning only 6 tips are visited.

```
> abmaxprune':: (MaxMin Integer) -> Integer -> Integer -> (Integer,[Int])
> abmaxprune' t a b| a==b = (a,[]) -- cutoff
>                 | otherwise = case t of
>                   (TipMax i v) -> (min (max v a) b,[i])
>                   (BranchMax ts) -> abmaxprunes' ts a b

> abmaxprunes' [] a b = (a,[])
> abmaxprunes' (t:ts) a b = case abmaxprunes' ts newa b of
>                       (v,indexes') -> (v,indexes++indexes')
>     where
>       (newa,indexes) = abminprune' t a b

> abminprune'::(MinMax Integer) -> Integer -> Integer -> (Integer,[Int])
> abminprune' t a b| a==b = (b,[]) --cutoff
>                 | otherwise = case t of
>                   (TipMin i v) -> (max (min v b) a,[i])
>                   (BranchMin ts) -> abminprunes' ts a b

> abminprunes' [] a b = (b,[])
> abminprunes' (t:ts) a b = case abminprunes' ts a newb of
>                       (v,indexes') -> (v,indexes++indexes')
>     where
>       (newb,indexes) = abmaxprune' t a b
```

The reason that \alpha-\beta pruning is VERY important in game playing
is that it allows one to search to almost double the ply that one could
with a basic min-max evaluation. This leads to a MUCH better game
playing performance!!!

Lesson8

Matching on steroids ....

I am continuing to use literate Haskell ...

Prolog, our next language, uses more powerful version of matching called
"unification" which is matching on steroids ...

Recall that the idea of MATCHING is that we are given an expression/term
(with variables), s, and a "template" (or a "pattern"), t, which is another expression/term
with variables which do not repeat. The expression/term, s , is said to match the
template/pattern t in case there is a substitution [t_1/x_1,...,t_n/x_n] of t such that

s == t[t_1/x_1,...,t_n/x_n]

The idea of UNIFICATION is completely symmetric: we are given two terms s_1 and s_2
and the objective is to find a substitution which makes them equal. Thus a
substitution [t_1/x_1,...,t_n/x_n] such that

s_1[t_1/x_1,...,t_n/x_n] = s_2[t_1/x_1,...,t_n/x_n]

such a substitution is called a UNIFIER (as applying it make the terms --syntactically-- equal).
However, there are many unifiers! But fortunately there is a "most general unifier" (upto
naming of variables). This is a substitution which is a unifier but has the additional
property that any other unifier can be obtained by substituting it.

Let us make that property a bit more explicit:

Suppose we are unifying s1 and s2 and we are given a unifier [t_1/x_1,...,t_n/x_n]
so that as above

t = s_1[t_1/x_1,...,t_n/x_n] = s_2[t_1/x_1,...,t_n/x_n].

A most general unifier [u_1/x_1,...,un/x_n] is a unifier, so

t' = s_1[u_1/x_1,...,un/x_n] =S_2[u_1/x_1,...,un/x_n]

but for the same there is a substitution of the kind that is the

but furthermore there is a substitution of t' which obtains t, that is there is a [v_1/y_1,...,v_m/y_m] so that

t = t'[v_1/y_1,...,v_m/y_m]

OK a bit of a mouthful HOWEVER to understand Prolog you really need to understand unification!! Of course, just as for matching, unification can fail: that is given two terms s_1 and s_2 it is quite possible that there is no unifier.

There is a huge amount of fundamental Computer Science work on unification (try googling Unification(computer science)): the main algorithms for unification (Alberto Martelli and Ugo Montanari) were only developed as recently as 1982 although the ideas go back further.

OK so we have to write a unification algorithm. Lets start with the datatype of expressions:

```
> data Exp f v  = Opn f [Exp f v]
>               | Var v
>   deriving (Show,Eq)
```

Of course we need the fold for expressions:

```
> foldExp:: (f -> [c] -> c) -> (v -> c) -> (Exp f v) -> c
> foldExp opn var (Var v) = var v
> foldExp opn var (Opn f args) = opn f (map (foldExp opn var) args)
```

A substitution for expressions with function type f and variable type v looks like:

```
> type Sub f v = (v,Exp f v)
> type Subs f v = [(v,Exp f v)]
```

Lets start with a warm up and program a single substitution:

```
> substitution:: Eq v => (Exp f v) -> (Sub f v) -> (Exp f v)
> substitution t (v,s) = foldExp Opn (\w -> if v==w then s else Var w) t
```

Note all this involves is substituting the leaves holding a variable equal to v with s otherwise leaving it alone.

So now lets program a whole substitution list ...
How do we do it well we can use a fold again!!!

```
> substitutions:: Eq v => (Exp f v) -> (Subs f v) -> (Exp f v)
> substitutions t subs =
>       foldExp Opn  (\w -> foldr  (\(v,s) t' -> if v==w then s else t') (Var w) subs) t
```

So the only subtely is that for each variable we must now run through the sbstitutions to see if we must replace the variable (we replace with the first available substitution always!).

Well that went pretty well ... here is an expression and a substitution list so you can try it out and check that it works!!! Invent your own test here ...

```
> e1 = Opn "add" [Opn "mult" [Var 1,Var 2],Var 1]
> e2 = Opn "mult" [Var 6,Var 7]
> e3 = Opn "add" [Var 1,Opn "mult" [Var 6,Var 7]]
> e4 = Opn "add" [Var 1,Opn "mult" [Var 1,Var 2]]
> e5 = Opn "add" [Var 9,Var 10]
> subs1 = [(1,e2),(2,e1)]
```

Now the unification algorithm starts with a "matching" step in which the trees are matched against each other and whenever one tree ends in a variable where the other tree continues one obtains a substitution. The substitution is only valid, however if it passes what is called "the occurs check" this check is that the substituted variable must not occur in the term which is to replace it (without the occurs check one gets an infinite tree!!).

We need the succeess or fail datatype:

```
> data SF  a = SS a | FF
>  deriving (Show,Eq)
>
> sfmap::(a -> b) -> (SF a) -> (SF b)
> sfmap f (SS a) = SS (f a)
> sfmap f FF = FF
>
> sflist:: [SF a] -> (SF [a])
> sflist = foldr (\x xs -> case (x,xs) of
>                          ((SS a),(SS as)) -> SS(a:as)
>                          _ -> FF)
>              (SS [])
>
> sfflatten:: (SF (SF a)) -> SF a
> sfflatten (SS x) = x
> sfflatten FF = FF
```

Now the occurs check. Note that v occuring in the expression which is the variable v is OK but no substitution is needed.

```
> occurs_check::(Eq f,Eq v) => (Sub f v) -> (SF (Subs f v))
> occurs_check (v,t)| t == (Var v) = SS []
```

```
>               | occurs v t = FF
>               | otherwise = SS [(v,t)]
>    where
>       occurs v = foldExp (\f args -> foldr (||) False args) (\w -> v==w)
```

Here is the matching step:

```
> matching::(Eq f,Eq v) => (Exp f v,Exp f v) -> (SF (Subs f v))
> matching ((Var v),t) = occurs_check (v,t)
> matching (s,(Var v)) = occurs_check (v,s)
> matching ((Opn f1 args1),(Opn f2 args2))| f1 == f2 && (length args1) == (length args2)
>       = sfmap concat (sflist (map matching (zip args1 args2)))
> matching _ = FF
```

This produces a substitution list however we need to go through the list tidy up the substitutions. We want a list of substitutions

[t_1/x_1,t_2/x_2,...,t_n/x_n]

such that for 1 =< i < j =< n the variable x_i does not occur in t_j. We can achieve this by substituting out (here "subout") the variable x_i by t_i in each t_j. However there are two things that can go wrong. First x_i could equal x_j in which case we retain the first substitution but replace the second by the substitution obtained by matching on the two right hand sides (as they must be made equal!). Second one can perform the substitution of x_i and the result can fail the occurs check ... in which case everything must fail. This gives:

```
> linearize:: (Eq v, Eq f) => (Subs f v) -> (SF (Subs f v))
> linearize [] = (SS [])
> linearize (sub:subs) = case subout sub subs of
>                          SS subs' -> SS (sub:subs')
>                          _ -> FF

> subout:: (Eq f,Eq v) => (Sub f v) -> (Subs f v) -> SF (Subs f v)
> subout s [] = SS []
> subout s@(x,t1) ((y,t2):rest)
>         | x==y = case matching (t1,t2) of
>                    SS msubs -> case subout s rest of
>                      SS subouts -> linearize (msubs++subouts)
>                      _ -> FF
>                    _ -> FF
>         | otherwise = case subout s rest of
>                     SS subouts  -> case (occurs_check (y,(substitution t2 s))) of
>                       SS subst -> SS (subst++subouts)
>                       _ -> FF
>                     _ -> FF
>
```

```
> unify t1 t2 = sfflatten ((sfmap linearize) (matching (t1,t2)))
```

Test the code try "unify ei ej" ...

Now this code is prety annoying as we have to repeatedly handle the case that a substitution fails.
As shall see shortly we can do this more cleanly using MONADS ...

Lesson9

Exceptions with monads ...

I am continuing to use literate Haskell ...

There is an important high-order class defined in Haskell called
Monad (which unfortunately in the prelude is now defined based
on having an "Applicative" which in turn is defined on having a
"Functor"!).

Let me go ahead and set up the success or fail type as a monad --
which is a bit complicated (in my humble opinion unnecessarily so!!) due
to these inter-dependencies which have been enforced by the prelude.
(This is perhaps a by-product of the industrialization of Haskell).

Let us not worry too much about what it all means at the momment but
rather try to see what the effect of having an exception monad for
programming. We will exemplify it on the code for unification ...

```
> data SF a = SS a | FF
>   deriving (Eq,Show)

> instance Functor SF where
>     fmap f FF = FF
>     fmap f (SS x) = SS (f x)

> instance Applicative SF where
>       pure x = SS x
>       FF <*> _ = FF
>       (SS f) <*> m = fmap f m

> instance Monad SF where
>       return x = SS x
>       (SS x) >>= k = k x
```

```
>       FF >>= _ = FF
```

Here is the rewritten sflist and the helper functions myhead and mytail:

```
> sflist:: [SF a] -> (SF [a])
> sflist [] = SS []
> sflist (a:as) = do a' <- a
>                    as' <- sflist as
>                    return (a':as')
```

The "do" syntax is what we must understand. We may read the last phrase of the code above as:

"do, get the successful part of a, assign it to a' now recursively get the successful part of as, assign it to as' now return (as successful) a':as'"

Note how the tracking of the failure condition (or the exception) is now automatically handled. This makes the code look a little neater ...

It will be useful to be able to extract the head and tail of a list. Note how we do so using the success or fail type to indicate when the extraction will fail:

```
> myhead::[a] -> (SF a)
> myhead [] = FF
> myhead (a:_) = SS a

> mytail::[a] -> (SF [a])
> mytail [] = FF
> mytail (_:as) = SS as
```

Now let us rewrite the unification program using the success or fail monad which we have now set up. First as before -- no change -- we need expressions and substitution:

```
> data Exp f v  = Opn f [Exp f v]
>               | Var v
>    deriving (Show,Eq)

> foldExp:: (f -> [c] -> c) -> (v -> c) -> (Exp f v) -> c
> foldExp opn var (Var v) = var v
> foldExp opn var (Opn f args) = opn f (map (foldExp opn var) args)

> type Sub f v = (v,Exp f v)
> type Subs f v = [(v,Exp f v)]
```

```
> substitution:: Eq v => (Exp f v) -> (Sub f v) -> (Exp f v)
> substitution t (v,s) = foldExp Opn (\w -> if v==w then s else Var w) t

> substitutions:: Eq v => (Exp f v) -> (Subs f v) -> (Exp f v)
> substitutions t subs =
>       foldExp Opn  (\w -> foldr  (\(v,s) t' -> if v==w then s else t') (Var w) subs) t

> e1 = Opn "add" [Opn "mult" [Var 1,Var 2],Var 1]
> e2 = Opn "mult" [Var 6,Var 7]
> e3 = Opn "add" [Var 1,Opn "mult" [Var 6,Var 7]]
> e4 = Opn "add" [Var 1,Opn "mult" [Var 1,Var 2]]
> e5 = Opn "add" [Var 9,Var 10]
> subs1 = [(1,e2),(2,e1)]
```

Here is our occurs check (also unchanged):

```
> occurs_check::(Eq f,Eq v) => (Sub f v) -> (SF (Subs f v))
> occurs_check (v,t)| t == (Var v) = SS []
>                   | occurs v t = FF
>                   | otherwise = SS [(v,t)]
>     where
>         occurs v = foldExp (\f args -> foldr (||) False args) (\w -> v==w)
```

Here is the matching function (note the last phrase):

```
> matching::(Eq f,Eq v) => (Exp f v,Exp f v) -> (SF (Subs f v))
> matching ((Var v),t) = occurs_check (v,t)
> matching (s,(Var v)) = occurs_check (v,s)
> matching ((Opn f1 args1),(Opn f2 args2))| f1 == f2
                                  && (length args1) == (length args2)
>         = do ssubs <- sflist (map matching (zip args1 args2))
>              return (concat ssubs)
>                                  | otherwise = FF
```

In the linearizing we can use the exception monad more significantly to our advantage:

```
> linearize:: (Eq v, Eq f) => (Subs f v) -> (SF (Subs f v))
> linearize [] = (SS [])
> linearize ssubs = do sub <- myhead ssubs
>                      subs <- mytail ssubs
>                      subouts <- subout sub subs
>                      return (sub:subouts)
```

In substituting out also is made neater using the exception monad:

```
> subout:: (Eq f,Eq v) => (Sub f v) -> (Subs f v) -> SF (Subs f v)
```

```
> subout :: (Eq t,Eq v) => (Sub t v) -> (Subs t v) -> SS (Subs t v)
> subout _ [] = SS []
> subout s@(x,t1) ((y,t2):rest)
>         | x==y =      do msubs <- matching (t1,t2)
>                         subouts <- subout s rest
>                         linearize (msubs++subouts)
>         | otherwise = do sub <- occurs_check (y,(substitution t2 s))
>                         subouts <- subout s rest
>                         return (sub ++ subouts)
```

This leaves the unify function itself:

```
> unify t1 t2 = do subs <- matching (t1,t2)
>                 linearize subs
```

Lessson 10

Monads here, monads there, monads everywhere!

I am continuing to use literate Haskell ...

Monads as we discussed are introduced using the class system.
Annoyingly they have been implemented to be dependent on "Applicative" and
"Functor". So let us walk through these various definitions and try to
explain them.

The Functor Class
==========

The simplest (higher-order) class is the Functor class:

```
class Functor m where
    fmap: (a -> b) -> (m a) -> (m b)
```

It is defined in the prelude and we shall let the definition there stand.

Here are some instances of this class:

(1) for lists (this in the prelude):

```
instance Functor [] where
    fmap f [] = []
    fmap f (a:as) = (f a):(fmap f as)
```

(2) for the success or fail datatype:

```
> data SF a = SS a | FF
>    deriving (Eq,Show)
>
> instance Functor SF where
>       fmap f FF = FF
>       fmap f (SS x) = SS (f x)
```

(3) For the datatype of binary trees:

```
> data Tree a = Leaf a
>             | Node (Tree a) (Tree a)
>    deriving (Eq,Show)
>
> instance Functor Tree where
>       fmap f (Leaf x) = Leaf (f x)
>       fmap f (Node t1 t2) = Node (fmap f t1) (fmap f t2)
```

(4) For the datatype of expressions:

```
> data Exp f v = Opn f [Exp f v]
>              | Var v

> instance Functor (Exp f) where
>       fmap f (Var x) = Var (f x)
>       fmap f (Opn g args) = Opn g (fmap (fmap f) args)
```

The functor thus does a replacement of the variables of an expression.

Note also the "currying" form of the type constructor. As defined
expressions -- the type constructor Exp -- has the "kind" type

```
    Exp:: * -> * -> *
```

so when f::* if follows that:

```
    Exp f:: * -> *
```

The latter is the kind type the functor class needs.

The Applicative class
=====================

This, unfortunately, is a class which is hard to justify and stands
between us and monads. Here is its definition:

```
 class (Functor m) => Applicative m where
```

```
pure:: a -> (m a)
(<*>):: (m (a -> b)) -> (m a) -> (m b)
```

The infix operation <*> is the applicative its intuitive idea is that one should be able to apply a function through a functor. So let give some examples:

(1) For lists:

```
instance Applicative [] where
  pure x = [x]
  (f:fs) <*> as = (fmap f as)++(fs <*> as)
```

For lists this takes a list of functions and applies it to a list of arguments by applying each f to each element of the list. We could write this as a list comprehension:

```
fs <*> xs = [ f x | f <- fs, x <- xs]
```

Or using the do syntax this is:

```
fs <*> xs = do f <- fs
               x <- xs
               return (f x)
```

However, list comprehension relies on having a monad so we cannot really do this ...

(2) For the success or fail datatype the applicative is quite straightforward:

```
> instance Applicative SF where
>       pure x = SS x
>       FF <*> _ = FF
>       (SS f) <*> m = fmap f m
```

One we have the exception monad defined we can express the applicative as:

```
fs <*> xs = do f <- fs
               x <- xs
               return (f x)
```

which is the same as for lists.

(3) For the datatype of binary trees the applicative is:

```
> instance Applicative Tree where
>     pure x = Leaf x
>     (Leaf f) <*> ts = fmap f ts
>     (Node t1 t2) <*> ts = Node (t1 <*> ts) (t2 <*> ts)
```

Again once we have the tree monad in place we may regard this, using the do syntax, as:

```
   fs <*> xs = do f <- fs
                  x <- xs
                  return (f x)
```

(4) For the datatype of expressions the applicative is:

```
> instance Applicative (Exp f) where
>     pure x = Var x
>     (Var g) <*> ts = fmap (\x -> g x) ts
>     (Opn f args) <*> ts = Opn f (fmap (\gs -> gs <*> ts) args)
```

The Monad class
==========

The way the monad class is set up in the prelude is as follows:

```
  class (Applicative m) => Monad m where
    return:: a -> (m a)
    (<*>):: (m a) -> (a -> m b) -> (m b)
```

we can now set up monads for lists, exceptions and trees:

(1) For lists this is already done in the prelude. It looks something like:

```
 instance Monad [] where
    return x = [x]
    xs >>= f = concat (fmap f xs)
```

we can now use the do syntax to write functions here is the pairing function:

```
> pairing:: [a] -> [b] -> [(a,b)]
> pairing as bs = do a <- as
>                    b <- bs
>                    return (a,b)
```

Recall the way of writing this as a list comprehension:

```
pairing as bs = [(a,b)| a <- as, b <- bs ]
```

The syntaxes are interchangeable.

(2) For the success or fail datatype the instance of a moad looks like:

```
> instance Monad SF where
>     return x = SS x
>     (SS x) >>= f = f x
>     FF >>= _ = FF
```

Here is the code for the function "sflist" using the do syntax:

```
> sflist:: [SF a] -> (SF [a])
> sflist [] = SS []
> sflist (a:as) = do a' <- a
>                   as' <- sflist as
>                   return (a':as')
```

Here is the banner example of quick sort in the do syntax:

```
> qs:: (Ord a) => [a] -> [a]
> qs [] = []
> qs (x:xs) = (qs (do x1 <- xs
>                     if x1 < x then return x1 else []))
>               ++ [x] ++
>           (qs (do x2 <- xs
>                   if x2 >= x then return x2 else []))
```

(3) For trees here is the instance of a monad:

```
> instance Monad Tree where
>     return x = Leaf x
>     (Leaf x) >>= f = f x
>     (Node t1 t2) >>= f = Node (t1 >>= f) (t2 >>= f)
```

Here is an example of the use of this monad: consider the problem of substituting the variable (or leaf) of a tree:

```
> subst:: (Eq v) => (v,Tree v) -> (Tree v) -> (Tree v)
> subst (v,s) t = do x <- t
>                   if x==v then s else (return x)
```

(4) Here is the monad instance for expressions:

```
> instance Monad (Exp f) where
>       return x = Var x
>       (Var x) >>= f = f x
>       (Opn g args) >>= f = Opn g (fmap (\arg -> arg>>=f) args)
```

Lets redo substitution but for expressions:

```
> substExp:: (Eq v) => (v,Exp f v) -> (Exp f v) -> (Exp f v)
> substExp (v,s) t = do x <- t
>                       if x==v then s else return x
```

Translating the "do" syntax
==========

The special "do" syntax for monads which makes them much more convenient to use is just syntax! The sysntax can be translated out as follows:

```
    do { r } = r
    do { x <- p ; C ; r } = p >>= q   where q x = do { C ; r }
    do { r' ; C ; r } = r' >>= \_ -> ( do { C ; r } )
```

Lesson11
Monads here, monads there, monads everywhere!

```
> import System.IO
```

More literate Haskell ...

The State Monad
==========

An important monad -- that it is usueful to know about -- is the state monad. To get the state monad going it is necessary to introduce the "state changing" type as a datatype:

```
> data (SM s a) = Sm (s -> (a,s))
```

There is only one constructor, Sm, so this means the type is essentially

s -> (a,s) i.e. a function which given a state produces a paired value
and new state. As the class system requires we have an explicit
data constructor.

A useful bit of code is the "running" or "application" of the monad type
on a state.

```
> runSM:: (SM s a) -> s -> (a,s)
> runSM (Sm h) s = h s
```

As before to get a monad going we need to introduce its curried kind
as a functor (recall SM:: * -> * -> * is a type constructor with two
arguments and the functor class wants a type constructor with just one
argument (SM s):: * -> *):

```
> instance Functor (SM s) where
>     fmap f (Sm h) = Sm (\s -> (\(a,s') -> (f a, s')) (h s))
```

Next one needs to define SM s as an applicative:

```
> instance Applicative (SM s) where
> --    pure:: a -> (SM s a)
>     pure a = Sm (\s -> (a,s))
> --    (<*>) ::(SM s (a->b)) -> (SM s a) -> (SM s b)
>     u <*> v
>           = Sm (\s -> (\(f,s') -> (\(a,s'') -> (f a, s'')
>                                ) (runSM v s')
>                   )(runSM u s))
```

The applicative here is the state change given by running the
state change mandated by u to obtain a function, f, and a new state, s',
then one uses this state, s', to run the state change mandated by , v,
to obtain the value, a, and a new state, s''. One returns
the state s'' paired with the evaluation of f a.

Once you have the monad this can be expressed as

```
    u <*> v =  do f <- u
                  a <- v
                  return (f a)
```

Finally, we can declare this state changing device as a monad:

```
> instance Monad (SM s) where
>     return a = Sm (\s -> (a,s))
>     (Sm f) >>= g = Sm (\s -> (\(a,s') -> runSM (g a) s') (f s))
```

Numbering the leaves of an expression:
==========

Here is an example of using the state monad to number the leaves
of an expresssion:

```
> data Exp f v = Var v
>              | Opn f [Exp f v]
>    deriving (Eq,Show)
```

Numbering the leaves of an expression

```
> number_exp :: (Exp a b) -> (SM Int (Exp a Int))
> number_exp (Var x)
>    = do n <- Sm (\n -> (n,n+1))
>         return (Var n)
> number_exp (Opn f args)
>    = do args' <- number_args args
>         return (Opn f args')

> number_args :: [(Exp a b)] -> (SM Int [(Exp a Int)])
> number_args [] = return []
> number_args (x:xs)
>    = do x' <- number_exp x
>         xs' <- number_args xs
>         return (x':xs')

> ex1 = runSM (number_exp (Opn "f" [Var "a",Var "b",Var"a"])) 0
```

Replacing variable names with numbers:
==========

A more sophisticated example: substituting the variables of an
expression with (new) numbers. This is useful if you are trying
to standardize the names of variables ...

```
> type ISubs v = [(v,Int)]

> data SF a = SS a | FF -- success or fail data type
>    deriving (Eq,Show)
```

The substitution of a string either replaces the string with a number
as specified in the list of substitutions or adds a substitution using
the least number still available ...

```
> subst:: Eq v => v -> (SM (ISubs v) Int)
```

```
> subst:: Eq v => v -> (SM (ISubs v) Int)
> subst str =
>   Sm (\subs -> case (insubs str subs) of
>                SS n -> (n,subs)
>                FF ->(next_free str 0 subs))
>    where
>       insubs:: Eq v => v -> (ISubs v) -> (SF Int)
>       insubs str [] = FF
>       insubs str ((str',n):subs)| str==str' = SS n
>                                 | otherwise = insubs str subs
>       next_free:: Eq v => v -> Int -> (ISubs v) -> (Int,ISubs v)
>       next_free str n [] = (n,[(str,n)])
>       next_free str n ((str',m):subs)
>                | n < m = (n,(str,n):((str',m):subs))
>                | otherwise = (\(m',subs') -> (m',((str',m):subs')))
>                                 (next_free str (n+1) subs)
```

So here the function "insubs" finds the substitution (the number which is to replace the variable) from a list of substitutions -- if it is there.
If it is not there then a new substituion must be generated: in this case "next_free" finds the least number no yet used for a substitution. It does this by looking through the substitution list (which is held in order) until it finds a number which has no been used.

Here then is code for substitution/replacement of the variables an expression by integers using the state monad ...

```
> subst_exp:: Eq v => (Exp f v) -> (SM (ISubs v) (Exp f Int))
> subst_exp (Var x) = do n <- subst x
>                        return (Var n)
> subst_exp (Opn f ts) = do ts' <- subst_list ts
>                           return (Opn f ts')
>  where
>        subst_list [] = return []
>        subst_list (t:ts) = do t' <- subst_exp t
>                               ts' <- subst_list ts
>                               return (t':ts')

> ex2 = runSM (subst_exp (Opn "f" [Var "a",Var "b",Var "a"])) []
```

Applying game moves
==========

In the checkers program you have to apply a move (which is a list of touch down coordinates of a piece) to a game state: this can be programmed as a state monad. Now you should not take this example as me advising you to

implement it in this manner. By using features which are more sophisticated may make your code harder to debug. However, this is a way to implement things and it does illustrate the utility of the state monad.

To statr with we need a datatype to represent the various sorts of pieces which inhabit a checkers board:

```
data PieceType = RedPiece | BlackPiece | RedKing | BlackKing | NoPiece
```

Now the idea is we need to write "apply_move":

```
apply_move::Move ->  GameState -> GameState
apply_move mv st | member mv (moves st) = snd (runSM (mover mv) st)
                 | otherwise = setmessage st "Illegal move!"
```

First we check that the move is legal by generating the legal moves using "move". If the move is legal we run the state monad on a function called "move" and extract the state (which is a Gamestate) from the result.

Now we must write the mover fuction and here we rely on the state monad:

```
mover:: Move -> (SM GameState ())
mover [end] = return ()
mover (m1:m2:ms) =
        do   pt <- remove_piece m1
             _  <- remove_piece (jumped m1 m2)
             () <- add_piece  pt m2
             mover (m2:ms)
```

You can actually do "apply_move" for the simple moves and jump moves all together: the trick is in the function "jumped". It should return the position jumped over ... but what if there is nothing to jump over? That is it was a simple move ... well you could return SF(Coord) and then modify the "remove_piece" function to do nothing when it recieves a fail (it can return any memebre of Piecetype say NoPiece).

Here we shall just boot this problem down the road (to you) and give the basic idea of the code:

```
remove_piece:: Coord -> SM(GameState,PieceType)
remove_piece xy = SM (piece_remover xy)
  where
    piece_remover:: Cood -> GameState -> (GameState,PieceType)
    piece_remover xy st| member xy (_redKings st)
          = (st{_redKings = remove xy (_redKings st)},RedKing)
                | member xy (_redPieces st)
```

```
        = ...

add_piece:: PieceType -> Coord -> SM(GameState,())
add_piece RedKing xy = Sm (\st -> (st{_redPieces = xy:(_redPieces st)},()))
...
```

Monadic I/O
==========

Haskell implements I/O using monads and the type IO a. It has various functions for I/O in particular:

```
putstr:: String -> IO ()
getLine:: IO String
```

The type () is the "unit type": it is an empty tuple so contains no information ... but is useful if you want to NOT convey any information!!

The I/O monad is builtin on the type IO: * -> * an this means one can use the do synatx to do some simple I/O. For example:

```
> myhead (x:xs) = SS x
> myhead _ = FF

> moodtest = do  putStr "Are you happy?\n"
>               ans <- getLine
>               case myhead ans of
>                   SS 'y' -> putStr "Hurrah!\n"
>                   SS 'n' -> putStr "Oh dear!\n"
>                   FF -> putStr "Not sure ... hmmm!\n"
```

There are lots of functions associated with I/O which you can explore ...