

Haskell Utilities

Brian Alliet

November 25, 2006

Contents

1	Math Functions	5
1.1	Matrix Functions	5
1.2	Number Theory Functions	7
1.3	Misc Math Functions	15
1.4	Bool Num Instance	16
2	Data Structures	17
2.1	Balanced Tree	17
2.2	Trie	25
2.3	Tree	27
2.4	Memory Buffer	29
3	Data Structure Utilities	32
3.1	Array Functions	32
3.2	List Functions	32
3.3	Maybe Functions	38
4	Monads	39
4.1	Failure Monad	39
4.2	Misc Monad Functions	40
5	Streams	41
5.1	Input Streams	41
5.2	Output Streams	44
5.3	Consumers	46
5.4	Producers	46
6	Text Processing and Parsers	47
6.1	CSV Library	47
6.2	Parsec Functions	49
6.3	HTML Parser	50
6.4	Html Utilities	53
6.5	Form Functions	56
7	Network Libraries	59
7.1	CURL Bindings	59

7.2	HTTP Protocol Utilities	63
7.3	URI Functions	64
7.4	Web Browser	67
7.5	Misc IO Functions	68
7.6	Foreign Function Interface Utilities	69
8	Type Level Magic	70
8.1	Type Level Naturals	70
8.2	Strongly Typed Heterogeneous Lists	72
9	Type-Safe Words	74
9.1	Introduction	74
9.2	Word Types	74
9.3	Operations	75
9.4	Standard Typeclass Instances	76
9.5	Advanced Operations	79
9.6	Type Aliases	81
10	Miscellaneous	82
10.1	Debugging Function	82
10.2	Misc Helper Functions	82
10.3	Operators	84

Introduction

This is a small (although growing) library of useful Haskell modules.

Chapter 1

Math Functions

1.1 Matrix Functions

The Hill cipher make heavy use of matrices. Matrix multiplication and inversion are key parts of the cipher. What follows is a small library of matrix functions for use with the Hill cipher (and possibly other ciphers in the future).

```
matrixMap :: (a → b) → [[a]] → [[b]]
matrixMap = map . map
```

`matrixMap` simply applies a given function to every element of the a matrix and returns the resultant matrix.

```
matrixAdd :: Num a ⇒ [[a]] → [[a]] → [[a]]
matrixAdd = zipWith (zipWith (+))
```

`matrixAdd` adds the corresponding elements of two matrices together.

```
matrixMul :: Num a ⇒ [[a]] → [[a]] → [[a]]
matrixMul a b = [[r `dotProd` c | c ← transpose b | r ← a]

matrixMulMod :: Integral a ⇒ a → [[a]] → [[a]] → [[a]]
matrixMulMod m a b = matrixMap (flip mod m) $ matrixMul a b
```

`matrixMul` multiplies two matrices. For each row of *a* the dot product is found for the entire row and each of the columns of *b*.

```
matrixMinor :: Num a ⇒ Int → Int → [[a]] → a
matrixMinor i j = matrixDet . skip i . map (skip j)
```

`matrixMinor` finds the minor for a given position in a matrix. The minor is calculated by finding the determinant (see `matrixDet`) of the matrix that results from removing the all elements in the same row or column as the given element.

```
matrixDet :: Num a => [[a]] -> a
matrixDet [] = 1
matrixDet m@(f:_) = f `dotProd` (head $ matrixCofactor m)
```

`matrixDet` finds the determinant of a square matrix. The determinant is calculated by multiplying each element of the first row by its cofactor.

```
matrixCofactor :: Num a => [[a]] -> [[a]]
matrixCofactor m = zipWith (map . (*)) signs
                    $ map (zipWith (*) signs)
                      $ [[matrixMinor r c m | c <- indices] | r <- indices]
  where
    indices = [0..length m-1]
    signs = cycle [1,-1]
```

`matrixCofactor` finds the cofactor matrix for a given matrix. The cofactor matrix is calculated by calculating the minor at each position in the matrix, inverting the sign of every other element in each row, then finally inverting the sign of every element in every other row.

```
matrixAdjoint :: Num a => [[a]] -> [[a]]
matrixAdjoint = transpose . matrixCofactor
```

`matrixAdjoint` finds the adjoint of a given matrix. The adjoint is calculated by simply transposing the cofactor matrix.

```
matrixInv :: Fractional a => [[a]] -> [[a]]
matrixInv m = matrixMap (/matrixDet m) $ matrixAdjoint m
```

`matrixInv` finds the inverse of a matrix. The inverse is calculated by multiplying every element of the adjoint of the matrix by $\frac{1}{\det(m)}$.

```
matrixInvMod :: Integral a => a -> [[a]] -> [[a]]
matrixInvMod x m = matrixMap (mulMod x $ multInvMod x d) a
  where d = matrixDet m
        a = matrixAdjoint m
```

`matrixInvMod` finds the inverse of a matrix (mod m). The inverse is calculated by multiplying (mod m) every element of the adjoint matrix by the multiplicative inverse of the determinant.

```
matrixInvertible :: Num a => [[a]] -> Bool
matrixInvertible m = matrixDet m /= 0

matrixInvertibleMod :: Integral a => a -> [[a]] -> Bool
matrixInvertibleMod x m = gcd (matrixDet m) x == 1
```

`matrixInvertible` and `matrixInvertibleMod` test to see if a matrix can be inverted.

```
matrixSquare :: [[a]] -> Bool
matrixSquare m = and $ map ((==length m).length) m
```

`matrixSquare` returns true if the given matrix is square.

```

toMatrix :: Int → Int → [a] → [[a]]
toMatrix w h l
  | length m == h = m
  | otherwise = error "toMatrix: data incorrect size"
  where m = chunkifyExact w l

```

toMatrix simply convertes a list to a matrix.

```

matrixIndex :: Eq a ⇒ a → [[a]] → Maybe (Int, Int)
matrixIndex e = matrixIndex' 0
  where
    matrixIndex' _ [] = Nothing
    matrixIndex' y (r:rs) = case elemIndex e r of
      Just x → Just (x,y)
      Nothing → matrixIndex' (y+1) rs

```

matrixIndex returns the (x,y) position in the matrix of the given element (or Nothing if it is not found).

1.2 Number Theory Functions

```

mulMod :: Integral a ⇒ a → a → a → a
mulMod a b c = (b * c) `mod` a

productMod :: Integral a ⇒ a → [a] → a
productMod a = foldl (mulMod a) 1

```

```

squareMod :: Integral a ⇒ a → a → a
squareMod a b = (b * b) `rem` a

```

```

addMod :: Integral a ⇒ a → a → a → a
addMod a b c = (b + c) `mod` a

negateMod :: Integral a ⇒ a → a → a
negateMod a b = negate b `mod` a

```

```

eqMod :: Integral a ⇒ a → a → a → Bool
eqMod a b c = b `mod` a == c `mod` a

```

mulMod, productMod, addMod, squareMod and negateMod perform the corresponding operations modulo a .

```

pow' :: (Num a, Integral b) => (a -> a -> a) -> (a -> a) -> a -> b -> a
pow' _ _ _ 0 = 1
pow' mul sq x' n' = f x' n' 1
  where
    f x n y
      | n == 1 = x `mul` y
      | r == 0 = f x2 q y
      | otherwise = f x2 q (x `mul` y)
    where
      (q,r) = quotRem n 2
      x2 = sq x

```

`pow'` implements the standard exponentiation by squaring algorithm.

```

pow :: (Num a, Integral b) => a -> b -> a
pow = pow' (*) (\x->x*x)

powMod :: Integral a => a -> a -> a -> a
powMod m = pow' (mulMod m) (squareMod m)

```

`pow` uses `pow'` to implement normal exponentiation. `powMod` implements exponentiation (mod `m`).

```

extEuclid :: Integral a => a -> a -> (a,a,a)
extEuclid a 0 = (1,0,a)
extEuclid a b = (y,x-y*q,thegcd)
  where (x,y,thegcd) = extEuclid b r
        (q,r) = quotRem a b

```

This is the Extended Euclid's Algorithm implemented as a recursive function. The correctness of this algorithm can be proven using induction.

FIXME Do the proof

```

multInvMod :: Integral a => a -> a -> a
multInvMod b n =
  if thegcd /= 1
  then error $ (show n) ++ " is not relatively prime"
  else x `mod` b -- ensure it is within the base
  where (x,_,thegcd) = extEuclid (n `mod` b) b

```

The Extended Euclid's Algorithm is used to find the multiplicative inverse of a number (mod `b`). `extEuclid` returns the gcd and `x` and `y` in $ax + by = gcd$. If the gcd is 1 (indicating that the given number is relatively prime and has a multiplicative inverse) then `x` is treated as the multiplicative inverse. `x` must be the multiplicative inverse because when we substitute in for `b` and `gcd` in the above equation we get $ax + b(base) = 1$. $b(base)$ equals 0 (mod `b`) so $ax = 1$.

```

crt :: Integral a => [(a,a)] -> a
crt congruencies = x `mod` mprod
  where
    (a,m) = unzip congruencies
    mprod = product m
    z = map (div mprod) m
    y = zipWith multInvMod m z
    x = sum $ zipWith3 (\b c d -> b*c*d) a y z

```

crt implements the Chinese Remainder Theorem

```

lazyPrimeSieve :: [Integer] -> [Integer]
lazyPrimeSieve [] = []
lazyPrimeSieve (x:xs) = x : (lazyPrimeSieve $ filter (\y -> y `rem` x /= 0) xs)

lazyPrimes :: [Integer]
lazyPrimes = 2 : lazyPrimeSieve [3,5..]

```

lazyPrimes is a list of all the prime numbers. It is generated using the Sieve of Eratosthenes (so it isn't very fast).

```

#ifdef __GLASGOW_HASKELL__
fastPrimeSieve :: Int -> [(Int,Bool)]
fastPrimeSieve limit = assocs $ runST run
  where
    sqrLimit = intSqrt limit
    run :: ST s (UArray Int Bool)
    run = do
      a <- newArray (0,limit) True :: ST s (STUArray s Int Bool)
      mapM_ (\x -> writeArray a x False) [0,1]
      mapM_ (\x -> do
        p <- readArray a x
        when (p) $ mapM_ (\y -> writeArray a y False) [x*2,x*3..limit]
      ) [2..sqrLimit+1]
      unsafeFreeze a >>= return
#endif

```

fastPrimeSieve is a fast Sieve of Eratosthenes. It generates lists of primes in large blocks rather than one at a time. It works well even for large values lazyPrimes is painfully slow for larger values). This uses an array containing a slot for every number from 0 to limit. For every prime it finds it marks off multiples of that prime in the array (so it eliminates a whole bunch of numbers in one shot).

```

factor :: Integral a => a -> [(a,a)]
factor = factorWithBase (map fromIntegral lazyPrimes)

```

```

factorWithBase :: Integral a => [a] -> a -> [(a,a)]
factorWithBase base x'
  | x' <= 0 = error "factorWithBase: can only factor numbers > 0"
  | otherwise = f x' 0 base
  where
    f 1 0 _ = []
    f x _ [] = [(x,1)]
    f x e ps@(p:ps')
      | r /= 0 = (p,e) : f x 0 ps'
      | otherwise = f q (e+1) ps
      where (q,r) = quotRem x p

```

factor factors an integer and returns a list of prime factors and exponents. factorWithBase uses a factor base other than the list of all primes. (However, everything in the factor base should still be prime). If x cannot be factored within the factor base the last “factor” will be whatever is left over to the power of 1.

factor 450 = [(2,1), (3,2), (5,2)]

```

allFactors :: Integral a => a -> [a]
allFactors n = [product $ zipWith (^) ps xs | xs <- positionPerms [[0..y] | y <- es]]
  where (ps,es) = unzip $ factor n

```

allFactors finds all the prime and composite factors of an integer.

```

factorWithinBase :: Integral a => [a] -> a -> Maybe [(a,a)]
factorWithinBase fb n
  | null fs = Just fs
  | (fst $ last fs) `elem` fb = Just fs
  | otherwise = Nothing
  where fs = factorWithBase fb n

```

factorWithinBase is just like factorWithBase except Nothing is returned if x could not be factored with only the factor base.

```

quadraticSieveFactor :: Integral a => [a] -> a -> Maybe (a,a)
quadraticSieveFactor fb n = listToMaybe
  $ map (\(x,y) -> let g = gcd (x-y) n in (g,n`div`g))
  $ filter (\(x,y) -> x /= y && not (eqMod n x (-y)))
  $ map (\(x,fs) ->
    (x,productMod n $ zipWith (^) fb [e`div`2 | e <- fs])
  )
  $ filter ((all even).snd)
  $ map (\row ->
    let (ns,fs) = unzip row
        in (productMod n ns, map sum $ transpose fs)
  )
  $ concat $ map (flip listPerms rows) [2..fblen`div` 2]

```

quadraticSieveFactor factors a large integer using a quadratic sieve. A factor base and the integer are given as arguments and if factorization is possible two factors are returned. Factorization is performed using the following steps:

1. Find possible rows - See below

2. Find all permutations of the rows - `listPerms` is used to find all permutations (starting with the smallest ones) of the rows.
3. Multiply each set of rows - `productMod` and `transpose` are used to multiply all the rows in each group together. This results in an integer and a list of exponents where the integer squared is equivalent to the exponents when applied to the factor base.
4. Find dependencies mod 2 - Next we filter out all rows who don't have linear dependencies mod 2
5. Take the square root of the exponents - Now we take the square root of the exponents (by dividing each one by two) and apply them back to the factor base to get a single integer.
6. Filter out unhelpful results - If $x \equiv \pm y \pmod n$ then it won't do us much good in factoring n so these results are discarded.
7. Find factors - Finally the remaining results are run through `gcd` to find a non-trivial of n and the two factors are returned.

```

where
  fblen = length fb
  maxRows = fblen + fblen `div` 2
  searchRange = [1..fromIntegral $ fblen*4]

  rows = take maxRows
        $ mapMaybe (\x → do
            let sq = squareMod n x
                when (sq == 0) Nothing
                fs ← factorWithinBase fb sq
                return (x, map snd fs)
            )
        $ [intSqrt (i*n) + j | i←searchRange, j←searchRange]

```

To find every possible row for the matrix above we search through numbers in the form $\sqrt{ni} + j$ looking for numbers whose square mod n is able to be factored using the factor base. A few limits are set (in terms of the size of the factor base) to limit the number of results to some sane size.

```

legendre :: Integral a ⇒ a → a → a
legendre a p
  | p < 3 = error "p isn't an odd prime"
  | x == p-1 = -1
  | otherwise = x
  where x = powMod p a ((p-1) `div` 2)

```

`legendre` returns the value of the legendre symbol $\left(\frac{n}{p}\right)$ as described in [Coh93].

```

sqrtModPrime :: (Random a, Integral a) => a -> a -> Maybe a
sqrtModPrime p y'
  | p == 2 = Just y
  | y == 0 = Just 0
  | otherwise = case p `mod` 4 of
    1 -> case p `mod` 8 of
      1 -> shanksTonelli p y
      5 ->
          if d == 1 then Just $ powMod p y (p `div` 8 + 1)
          else if d == p - 1 then Just $ mulMod p
              (mulMod p y 2) (powMod p (mulMod p y 4) (p `div` 8))
          else Nothing
          where d = powMod p y (p `div` 4)
      _ -> error "sqrtModPrime: p isn't prime"
    3 ->
          if mulMod p x x == y then Just x else Nothing
          where x = powMod p y ((p+1) `div` 4)
      _ -> error "sqrtModPrime: p isn't prime"
  where y = y' `mod` p

```

`sqrtModPrime` find the square root of $y \pmod{p}$ where p is a prime number using the methods described in [Coh93]

FIXME: Talk about this some more... proofs?

```

sqrtModPrimePower :: (Random a, Integral a) => a -> a -> a -> Maybe a
sqrtModPrimePower p k a
  | k <= 0 = error "sqrtModPrimePower: k <= 0"
  | k == 1 = sqrtModPrime p a
  | p == 2 && k == 2 =
      case a `mod` 4 of
        0 -> Just 0 -- FIXME: actually [0,2]
        1 -> Just 1
        _ -> Nothing
  | otherwise = do
      when (p == 2 && a `mod` 8 /= 1) Nothing
      x0 <- sqrtModPrimePower p (k-1) a
      when (x0 == 0) Nothing
      let
          p' = p^(k-1)
          b = (x0*x0 - a) `div` p'
          y0 = mulMod p' (-b) (multInvMod p' (if p == 2 then x0 else 2*x0))
      return $ x0 + mulMod (p^k) p' y0

```

`sqrtModPrimePower` finds the square root of $a \pmod{p^a}$ where p is a prime number.

FIXME: This is still broken, need to find a book on this

```

shanksTonelli :: (Random a, Integral a) => a -> a -> Maybe a
shanksTonelli p a =
  iter
    z           — y ← z
    e           — r ← e
    (mulMod p a (powMod p x' 2)) — b ← ax^2
    (mulMod p a x')           — x ← ax
  where
    (z,_) = findZ $ mkStdGen 42
    (e,q) = find2km (p-1)
    x' = powMod p a (q`div`2)

  iter _ _ 1 x = Just x — done
  iter y r b x =
    case find (λm → powMod p b (2^m) == 1) [0..r-1] of
      Nothing → Nothing — isn't a square
      Just m → iter
        y'
        m           — r ← m
        (mulMod p b y') — b ← by
        (mulMod p x t) — x ← xt
        where
          t = powMod p y (2^(r-m-1)) — t ← y^(2^(r-m-1))
          y' = powMod p t 2           — y ← t^2 -}

  findZ g = if legendre n p == -1 then (powMod p n q,g') else findZ g'
  where (n,g') = randomR (2,p-1) g

```

shanksTonelli implements the Shanks-Tonelli algorithm as described in [Coh93]. This is a direct conversion of the algorithm description in the book to Haskell. (Which is why it maintains its imperative-like style).

```

sqrtMod :: (Random a, Integral a) => a -> a -> [a]
sqrtMod m x
  | m ≤ 0 = error "Brianweb.Math.sqrtMod: m must be ≥ 1"
  | m == 2 = [x `mod` 2] — FIXME: HACK
  | m == 4 && x `mod` 4 == 0 = [0,2] — FIXME: HACK
  | otherwise =
    case filter ((/=0).snd) $ factor m of
      [] → [0] — only happens if m == 1
      [(p,k)] →
        case sqrtModPrimePower p k x of
          Just 0 → [0]
          Just r → [r,negateMod (p^k) r]
          Nothing → []
    ps → nub
      $ map crt
      $ positionPerms
      $ map (λ(p,k) → map (λr → (r,p^k)) $ sqrtMod (p^k) x) ps

```

sqrtMod finds the square root of $x \pmod m$ as described in [TW02]

FIXME: This is still broke

```

euler :: Integral a => a -> a
euler x = numerator
  $ (fromIntegral x)
  * (product $ map ((1-).(1%)) $ [fst y|y←factor x,snd y ≠ 0])

```

euler is Eulers ϕ -function as described in [TW02]

```

millerRabinPrimality :: Integer → Integer → Bool
millerRabinPrimality n a
  | a ≤ 1 || a ≥ n-1 =
    error $ "millerRabinPrimality: a out of range ("
      ++ show a ++ " for " ++ show n ++ ")"
  | n < 2 = False
  | even n = False
  | b0 == 1 || b0 == n' = True
  | otherwise = iter (tail b)
  where
    n' = n-1
    (k,m) = find2km n'
    b0 = powMod n a m
    b = take (fromIntegral k) $ iterate (squareMod n) b0
    iter [] = False
    iter (x:xs)
      | x == 1 = False
      | x == n' = True
      | otherwise = iter xs

```

`millerRabinPrimality` implements the Miller-Rabin Primality Test as described in [TW02]. `b0` is b_0 and `b` represents the whole array of b s. Each b is generated by squaring the last b . `iter` runs a single iteration. If b is 1 then the number is classified as composite, if b is 1 the number is classified as prime, otherwise, the next iteration is run. If we run out of b s then the number is classified as composite.

```

probablyPrime :: (RandomGen a) ⇒ Integer → a → (Bool,a)
probablyPrime p
  | p ≤ smallPrime = λr → (smallCheck p,r)
  | otherwise = loop 10
  where
    loop :: RandomGen a ⇒ Int → a → (Bool,a)
    loop 0 rg = (True,rg)
    loop n rg =
      if millerRabinPrimality p r
        then loop (n-1) rg'
        else (False,rg')
      where (r,rg') = randomR (2,p-2) rg
#ifdef __GLASGOW_HASKELL__
    smallPrime = 100000
    smallCheck x = binarySearch (fromIntegral x) a
    where
      l = map fst $ filter snd
        $ fastPrimeSieve (fromIntegral smallPrime)
      a = array (0,length l-1) $ zip [0..] l
#else
    smallPrime = 2000
    smallCheck x = x `sortedElem` lazyPrimes
#endif

```

`probablyPrime` returns true if p it is very likely that p is prime. For small integers the Sieve of Eratosthenes is used to test primality. The Miller-Rabin test is done 10 times for larger integers.

```

getPrime :: (RandomGen a) ⇒ Int → a → (Integer,a)
getPrime n rg0 = if p then (r,rg2) else getPrime n rg2
  where
    (r,rg1) = randomR (2^(n-1),2^n-1) rg0
    (p,rg2) = probablyPrime r rg1

```

`getPrime` gets a random n -bit prime number using the Miller-Rabin test.

```
find2km :: Integral a => a -> (a,a)
find2km n = f 0 n
  where
    f k m
      | r == 1 = (k,m)
      | otherwise = f (k+1) q
      where (q,r) = quotRem m 2
```

`find2km` finds two numbers (k and m) such that $2^k m = n$ and m is odd.

```
order :: Integral a => a -> a -> a
order m a'
  | gcd a m == 1 = head $ [x|x<-allFactors (euler m),powMod m a x == 1]
  | otherwise = error "gcd a m /= 1"
  where a = a' `mod` m
```

`order` computes the order on a modulo m .

```
isPrimitiveRoot :: Integral a => a -> a -> Bool
isPrimitiveRoot m a'
  | gcd a m == 1 = order m a == euler m
  | otherwise = False
  where a = a' `mod` m
```

`isPrimitiveRoot` returns true if a is a primitive root modulo m .

```
allPrimitiveRoots :: Integral a => a -> [a]
allPrimitiveRoots m =
  case find (isPrimitiveRoot m) [1..m-1] of
    Just x -> [powMod m x k | k <- [1..euler m-1], gcd k (euler m) == 1]
    Nothing -> []
```

`allPrimitiveRoots` returns all primitive roots modulo m .

1.3 Misc Math Functions

```
dotProd :: Num a => [a] -> [a] -> a
dotProd = (sum.) . zipWith (*)
```

`dotProd` finds the dot product of two vectors. This is used for some matrix operation and for language statistics.

```
average :: Fractional a => [a] -> a
average = average' 0 0
  where
    average' n s [] = s / n
    average' n s (x:xs) = average' (n+1) (s+x) xs
```

average finds the average of all the values in a list.

```
intSqrt :: Integral a => a -> a
intSqrt n
  | n < 0 = error "Brianweb.Math.intSqrt: negative n"
  | otherwise = f n
  where
    f x = if y < x then f y else x
          where y = (x + (n `quot` x)) `quot` 2
```

intSqrt finds the floor of the square root of an integer using Newton's Iteration.

1.4 Bool Num Instance

This module brings Bool into the Num class. It behaves like a 1-bit binary number. Addition and subtraction are mapped to `is /=` and multiplication to `and`.

```
instance Num Bool where
  (+) = (/=)
  (-) = (/=)
  (*) = (&&)
  negate = id
  abs = id
  signum False = 0
  signum True = 1
  fromInteger = toEnum . fromIntegral
```

Chapter 2

Data Structures

2.1 Balanced Tree

2.1.1 Introduction

This is a simple Balanced Tree data structure based on [Ada93, DFM, IBT].

2.1.2 Data Declarations

```
data Map k v = E | L !k v | B !k v !Int !(Map k v) !(Map k v)
deriving Show
```

A balanced tree is either empty (E), a leaf L, or a branch (B). A leaf could also be treated as a branch with empty children but having a separate constructor is more efficient. However, the library will still work even if leaves are created inefficiently.

```
type Set a = Map a ()
type List a = Map () a
```

A balanced tree can also be used a set (a balanced tree with no value) or a list (a balanced tree with no key).

```
instance (Eq k, Eq v) => Eq (Map k v) where
  (==) = equals
```

With keys and values in the Eq class Map's can also be tested for equality,

2.1.3 Balancing Code

```
w :: Int
w = 4
```

w is the maximum ratio the two leaves before balancing is performed.

```
rotateL1, rotateL2, rotateR1, rotateR2 ::
  k -> v -> Map k v -> Map k v -> Map k v

rotateL1 b b' a (B d d' _ c e) = branch d d' (branch b b' a c) e
rotateL1 _ _ _ _ = impossible

rotateR1 d d' (B b b' _ a c) e = branch b b' a (branch d d' c e)
rotateR1 _ _ _ _ = impossible
```

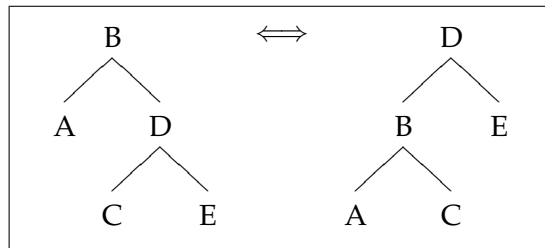


Figure 2.1: Single Rotation

```
rotateL2 b b' a (B f f' _ (B d d' _ c e) g) = branch d d' (branch b b' a c) (branch f f' e g)
rotateL2 b b' a (B f f' _ (L d d') g)       = branch d d' (branch b b' a E) (branch f f' E g)
rotateL2 _ _ _ _ = impossible
```

rotateL2 rotates a tree to the left as shown below.

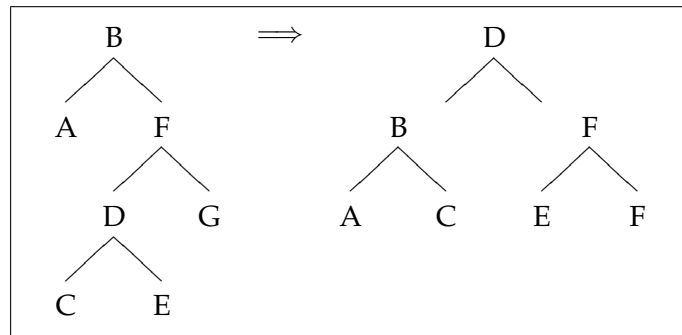


Figure 2.2: Double Left Rotation

```
rotateR2 f f' (B b b' _ a (B d d' _ c e)) g = branch d d' (branch b b' a c) (branch f f' e g)
rotateR2 f f' (B b b' _ a (L d d')) g       = branch d d' (branch b b' a E) (branch f f' E g)
rotateR2 _ _ _ _ = impossible
```

rotateR2 rotates a tree to the right as shown below.

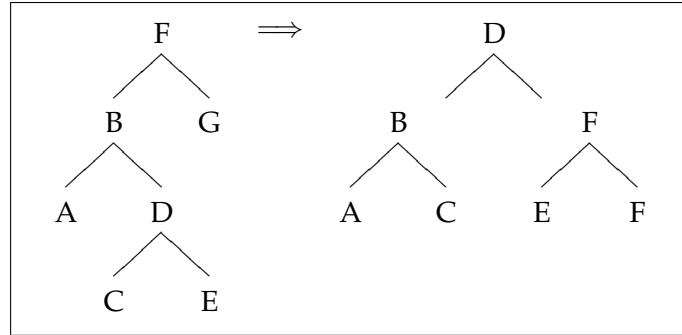


Figure 2.3: Double Right Rotation

```
branch :: k -> v -> Map k v -> Map k v -> Map k v
branch k v E E = L k v
branch k v l r = B k v (1 + size l + size r) l r
```

branch creates a branch from a key, a value, and two children, correctly calculating the size field.

```
balance :: k -> v -> Map k v -> Map k v -> Map k v
balance k v l r
  | ln + rn < 2 = branch k v l r
  | rn > ln*w = case r of
    B _ _ rl rr
      | size rl < size rr -> rotateL1 k v l r
      | otherwise         -> rotateL2 k v l r
    _ -> impossible
  | ln > rn*w = case l of
    B _ _ ll lr
      | size lr < size ll -> rotateR1 k v l r
      | otherwise         -> rotateR2 k v l r
    _ -> impossible
  | otherwise = branch k v l r
where
  ln = size l
  rn = size r
```

balance creates a branch from a key, value, and two children, balancing if necessary.

```
concat3 :: Ord k => k -> v -> Map k v -> Map k v -> Map k v — FIXME: Ord constraint
concat3 k v E r = insert k v r
concat3 k v l E = insert k v l
concat3 k v (L k' v') r = insert k v (insert k' v' r)
concat3 k v l (L k' v') = insert k v (insert k' v' l)
concat3 k v l@(B k1 v1 s1 l1 r1) r@(B k2 v2 s2 l2 r2)
  | w*s1 < s2 = balance k2 v2 (concat3 k v l l2) r2
  | w*s2 < s1 = balance k1 v1 l1 (concat3 k v r1 r)
  | otherwise = branch k v l r
```

```

concat :: Ord k => Map k v -> Map k v -> Map k v — FIXME: Ord constraint
concat E r = r
concat l E = l
concat (L k v) r = insert k v r
concat l (L k v) = insert k v l
concat l@(B k1 v1 s1 l1 r1) r@(B k2 v2 s2 l2 r2)
  | w*s1 < s2 = balance k2 v2 (concat l l2) r2
  | w*s2 < s1 = balance k1 v1 l1 (concat r1 r)
  | otherwise = (uncurry balance) (minPair r) l (deleteMin r)

```

2.1.4 Public API

```

size :: Map k v -> Int
size E = 0
size (L _ _) = 1
size (B _ _ n _ _) = n

null :: Map k v -> Bool
null E = True
null (B _ _ n _ _) | n == 0 = True
null _ = False

```

```

empty :: Map k v
empty = E

leaf :: k -> v -> Map k v
leaf = L

```

```

insert :: Ord k => k -> v -> Map k v -> Map k v
insert = insertAccum (const id)

```

```

insertAccum :: Ord k => (v -> v -> v) -> k -> v -> Map k v -> Map k v
insertAccum _ k' v' E = L k' v'
insertAccum c k' v' l@(L k v) = case compare k' k of
  LT -> branch k' v' E l
  GT -> branch k' v' l E
  EQ -> L k (c v v')
insertAccum c k' v' (B k v s l r) = case compare k' k of
  LT -> balance k v (insertAccum c k' v' l) r
  GT -> balance k v l (insertAccum c k' v' r)
  EQ -> B k (c v v') s l r

```

```

insertAccum' :: Ord k => (v -> v -> v) -> k -> v -> Map k v -> Map k v
insertAccum' _ k' v' E = L k' v'
insertAccum' c k' v' l@(L k v) = case compare k' k of
  LT -> branch k' v' E l
  GT -> branch k' v' l E
  EQ -> let v'' = c v v' in v'' `seq` L k v''
insertAccum' c k' v' (B k v s l r) = case compare k' k of
  LT -> balance k v (insertAccum' c k' v' l) r
  GT -> balance k v l (insertAccum' c k' v' r)
  EQ -> let v'' = c v v' in v'' `seq` B k v'' s l r

```

```

delete :: Ord k => Map k v -> k -> Map k v
delete E _ = E
delete l@(L k _) k'
  | k == k' = E
  | otherwise = l
delete (B k v _ l r) k' = case compare k' k of
  LT -> balance k v (delete l k') r
  GT -> balance k v l (delete r k')
  EQ
    | size l > size r -> (uncurry balance) (maxPair l) (deleteMax l) r
    | otherwise        -> (uncurry balance) (minPair r) l (deleteMin r)

```

```

{-# SPECIALIZE lookup :: Ord k => k -> Map k v -> Maybe v #-}
lookup :: (MonadFail m, Ord k) => k -> Map k v -> m v
lookup _ E = mfail "BalancedTree.lookup: key not found"
lookup k' (L k v)
  | k == k' = return v
  | otherwise = mfail "BalancedTree.lookup: key not found"
lookup k' (B k v _ l r) = case compare k' k of
  LT -> k' `lookup` l
  GT -> k' `lookup` r
  EQ -> return v

member :: Ord k => k -> Map k v -> Bool
member k bt = isJust (k `lookup` bt)

contains :: Ord k => Map k v -> k -> Bool
contains = flip member

```

```

pairAt :: Map k v -> Int -> (k,v)
pairAt (L k v) 0 = (k,v)
pairAt (B k v _ l r) i = case compare i (size l) of
  LT -> pairAt l i
  GT -> pairAt r (i - size l - 1)
  EQ -> (k,v)
pairAt _ _ = error "pairAt: index out of bounds"

valueAt :: Map k v -> Int -> v
valueAt bt i = snd (pairAt bt i)

keyAt :: Map k v -> Int -> k
keyAt bt i = fst (pairAt bt i)

```

```

insertAt :: Int -> v -> List v -> List v
insertAt i v E = case i of
  0 -> L () v
  _ -> error "insertAt: index out of bounds"
insertAt i v' l@(L _ _) = case i of
  0 -> branch () v' E l
  1 -> branch () v' l E
  _ -> error "insertAt: index out of bounds"
insertAt i v' (B k v _ l r)
  | i <= (size l) = balance k v (insertAt i v' l) r
  | otherwise = balance k v l (insertAt (i - size l - 1) v' r)

```

```

indexOfKey :: Ord k => Map k v -> k -> Maybe Int
indexOfKey = indexOfKey' 0
  where
    indexOfKey' _ E _ = Nothing
    indexOfKey' x (L k _) k'
      | k == k' = Just x
      | otherwise = Nothing
    indexOfKey' x (B k _ _ l r) k' = case compare k' k of
      LT -> indexOfKey' x l k'
      GT -> indexOfKey' (x + size l + 1) r k'
      EQ -> Just (x + size l)

indexOfValue :: Eq v => Map k v -> v -> Maybe Int
indexOfValue bt v = elemIndex v (values bt)

```

```

reverseLookup :: Eq v => Map k v -> v -> Maybe k
reverseLookup bt v = indexOfValue bt v >>= return.keyAt bt

```

```

minPair :: Map k v -> (k,v)
minPair E = error "minPair: empty tree"
minPair (L k v) = (k,v)
minPair (B k v _ E _) = (k,v)
minPair (B _ _ _ l _) = minPair l

maxPair :: Map k v -> (k,v)
maxPair E = error "maxPair: empty tree"
maxPair (L k v) = (k,v)
maxPair (B k v _ _ E) = (k,v)
maxPair (B _ _ _ _ r) = maxPair r

```

```

deleteMin :: Map k v -> Map k v
deleteMin E = impossible
deleteMin (L _ _) = E
deleteMin (B _ _ _ E r) = r
deleteMin (B k v _ l r) = balance k v (deleteMin l) r

deleteMax :: Map k v -> Map k v
deleteMax E = impossible
deleteMax (L _ _) = E
deleteMax (B _ _ _ l E) = l
deleteMax (B k v _ l r) = balance k v l (deleteMax r)

```

```

minKey,maxKey :: Map k v -> k
minKey = fst . minPair
maxKey = fst . maxPair

minValue,maxValue :: Map k v -> v
minValue = snd . minPair
maxValue = snd . maxPair

```

```

foldr :: (k → v → a → a) → a → Map k v → a
foldr _ z E = z
foldr f z (L k v) = f k v z
foldr f z (B k v _ l r) = foldr f (f k v (foldr f z r)) l

foldl :: (a → k → v → a) → a → Map k v → a
foldl _ z E = z
foldl f z (L k v) = f z k v
foldl f z (B k v _ l r) = foldl f (f (foldl f z l) k v) r

```

```

treeFold :: (k → v → a → a → a) → a → Map k v → a
treeFold _ z E = z
treeFold f z (L k v) = f k v z z
treeFold f z (B k v _ l r) = f k v (treeFold f z l) (treeFold f z r)

```

```

keys :: Map k v → [k]
keys = foldr (λk _ r → k:r) []

values :: Map k v → [v]
values = foldr (const (:)) []

assocs :: Map k v → [(k,v)]
assocs = foldr (λk v → ((k,v):)) []

```

```

#ifdef __GLASGOW_HASKELL__
{-# RULES
"assocs"  [~1] forall bt. assocs bt = build (λc n → foldr (λk v → c (k,v)) n bt)
"keys"    [~1] forall bt. keys  bt = build (λc n → foldr (λk _ → c k    ) n bt)
"values"  [~1] forall bt. values bt = build (λc n → foldr (λ_ v → c v    ) n bt)
#-}
#endif

```

```

insertAssocsAccum :: Ord k ⇒ (v → v → v) → [(k,v)] → Map k v → Map k v
insertAssocsAccum c = flip (P.foldl (λbt (k,v) → insertAccum c k v bt))

insertAssocsAccum' :: Ord k ⇒ (v → v → v) → [(k,v)] → Map k v → Map k v
insertAssocsAccum' c = flip (P.foldl (λbt (k,v) → insertAccum' c k v bt))

insertAssocs :: Ord k ⇒ [(k,v)] → Map k v → Map k v
insertAssocs = insertAssocsAccum (const id)

```

```

fromAssocsAccum :: Ord k ⇒ (v → v → v) → [(k,v)] → Map k v
fromAssocsAccum f xs = insertAssocsAccum f xs empty

fromAssocsAccum' :: Ord k ⇒ (v → v → v) → [(k,v)] → Map k v
fromAssocsAccum' f xs = insertAssocsAccum' f xs empty

fromAssocs :: Ord k ⇒ [(k,v)] → Map k v
fromAssocs xs = insertAssocs xs empty

fromList :: Ord k ⇒ [(k,v)] → Map k v
fromList = fromAssocs

setFromList :: Ord k ⇒ [k] → Set k
setFromList xs = fromList [(x, ()) | x ← xs]

```

```

split :: Ord k => Map k v -> k -> (Map k v, Maybe v, Map k v)
split E _ = (E,Nothing,E)
split l@(L k v) k' = case compare k k' of
  LT -> (l,Nothing,E)
  GT -> (E,Nothing,l)
  EQ -> (E,Just v,E)
split (B k v _ l r) k' = case compare k k' of
  LT -> let (lt,eq,gt) = split r k' in (concat3 k v l lt,eq,gt)
  GT -> let (lt,eq,gt) = split l k' in (lt,eq,concat3 k v gt r)
  EQ -> (l,Just v,r)

```

```

unionAccum :: Ord k => (v -> v -> v) -> Map k v -> Map k v -> Map k v
unionAccum _ E r = r
unionAccum _ l E = l
unionAccum c (L k v) r = insertAccum (flip c) k v r
unionAccum c l (L k v) = insertAccum c k v l
unionAccum c lr' (B k v _ l r) = concat3 k (maybe v (flip c v) v') lu ru
  where
    (l',v',r') = split lr' k
    lu = unionAccum c l' l
    ru = unionAccum c r' r

```

```

union :: Ord k => Map k v -> Map k v -> Map k v
union = unionAccum (const id)

```

```

difference :: Ord k => Map k v -> Map k v -> Map k v
difference E _ = E
difference l E = l
difference l (L k _) = delete l k
difference lr' (B k _ _ l r) = concat ld rd
  where
    (l',_,r') = split lr' k
    ld = l' `difference` l
    rd = r' `difference` r

```

```

intersection :: Ord k => Map k v -> Map k v -> Map k v
intersection = intersectionAccum (const id)

```

```

intersectionAccum :: Ord k => (v -> v -> v) -> Map k v -> Map k v -> Map k v
intersectionAccum _ E _ = E
intersectionAccum _ _ E = E
intersectionAccum c (L k v) r = case k `lookup` r of
  Just v' -> L k (c v v')
  Nothing -> E
intersectionAccum c l (L k v) = case k `lookup` l of
  Just v' -> L k (c v' v)
  Nothing -> E
intersectionAccum c lr' (B k v _ l r) = case v' of
  Just v'' -> concat3 k (c v'' v) li ri
  Nothing -> concat li ri
  where
    (l',v',r') = split lr' k
    li = intersectionAccum c l' l
    ri = intersectionAccum c r' r

```

```

map :: (k → v → v') → Map k v → Map k v'
map _ E = E
map f (L k v) = L k (f k v)
map f (B k v s l r) = B k (f k v) s (map f l) (map f r)

```

```

filter :: Ord k ⇒ (k → v → Bool) → Map k v → Map k v — FIXME
filter _ E = E
filter f l@(L k v)
  | f k v = l
  | otherwise = E
filter f (B k v _ l r)
  | f k v = concat3 k v fl fr
  | otherwise = concat fl fr
  where
    fl = filter f l
    fr = filter f r

```

```

partition :: Ord k ⇒ (k → v → Bool) → Map k v → (Map k v, Map k v) — FIXME
partition _ E = (E, E)
partition f l@(L k v)
  | f k v = (l, E)
  | otherwise = (E, l)
partition f (B k v _ l r)
  | f k v = (concat3 k v pll prl, concat plr prr)
  | otherwise = (concat pll prl, concat3 k v plr prr)
  where
    (pll, plr) = partition f l
    (prl, prr) = partition f r

```

```

equals :: (Eq k, Eq v) ⇒ Map k v → Map k v → Bool
equals x y = size x == size y && assocs x == assocs y

```

2.2 Trie

2.2.1 Introduction

This is a trie.

2.2.2 Data Declarations

```

data Trie k v = Trie ![k] !(Maybe v) !(BT.Map k (Trie k v))
  deriving Show

```

2.2.3 Public API

```
empty :: Trie k v
empty = Trie [] Nothing BT.empty

size :: Trie k v → Int
size (Trie _ mv bt) = BT.foldr (λ_ x → (+size x)) (maybe 0 (const 1) mv) bt
```

```
commonPrefix :: Eq a ⇒ [a] → [a] → ([a], [a], [a])
commonPrefix (x:xs) (y:ys) | x == y = let (p, xs', ys') = commonPrefix xs ys in (x:p, xs', ys')
commonPrefix xs ys = ([], xs, ys)

withoutPrefix :: Eq a ⇒ [a] → [a] → Maybe [a]
withoutPrefix (x:xs) (y:ys) | x == y = withoutPrefix xs ys
withoutPrefix [] ys = Just ys
withoutPrefix _ _ = Nothing
```

```
insertAccum :: Ord k ⇒ (v → v → v) → Trie k v → [k] → v → Trie k v
insertAccum c (Trie k mv children) k' v' = case commonPrefix k k' of
  (_, [], []) → Trie k (Just (maybe v' (λv → c v v') mv)) children
  (_, [], (x:xs)) → Trie k mv $ BT.insertAccum
    (λold _ → insertAccum c old xs v')
    x (Trie xs (Just v') BT.empty) children
  (cp, x:xs, []) → Trie cp (Just v') $ BT.leaf x (Trie xs mv children)
  (cp, x:xs, y:ys) → Trie cp Nothing $ BT.fromAssocs
    [(x, Trie xs mv children), (y, Trie ys (Just v') BT.empty)]
```

```
insertAccum' :: Ord k ⇒ (v → v → v) → Trie k v → [k] → v → Trie k v
insertAccum' c (Trie k mv children) k' v' = case commonPrefix k k' of
  (_, [], []) → let v'' = maybe v' (λv → c v v') mv in v'' 'seq' Trie k (Just v'') children
  (_, [], (x:xs)) → Trie k mv $ BT.insertAccum
    (λold _ → insertAccum c old xs v')
    x (Trie xs (Just v') BT.empty) children
  (cp, x:xs, []) → Trie cp (Just v') $ BT.leaf x (Trie xs mv children)
  (cp, x:xs, y:ys) → Trie cp Nothing $ BT.fromAssocs
    [(x, Trie xs mv children), (y, Trie ys (Just v') BT.empty)]
```

```
insert :: Ord k ⇒ Trie k v → [k] → v → Trie k v
insert = insertAccum (const id)

lookup :: Ord k ⇒ Trie k v → [k] → Maybe v
lookup (Trie k mv children) k' = case withoutPrefix k k' of
  Nothing → Nothing
  Just [] → mv
  Just (x:xs) → x 'BT.lookup' children >>= flip lookup xs
```

```
insertAssocAccum :: Ord k ⇒ (v → v → v) → Trie k v → [(k, v)] → Trie k v
insertAssocAccum c = P.foldl (λbt (k, v) → insertAccum c bt k v)

insertAssocAccum' :: Ord k ⇒ (v → v → v) → Trie k v → [(k, v)] → Trie k v
insertAssocAccum' c = P.foldl (λbt (k, v) → insertAccum' c bt k v)

insertAssoc :: Ord k ⇒ Trie k v → [(k, v)] → Trie k v
insertAssoc = insertAssocAccum (const id)
```

```
fromAssocAccum :: Ord k => (v -> v -> v) -> [(k,v)] -> Trie k v
fromAssocAccum f = insertAssocAccum f empty
```

```
fromAssocAccum' :: Ord k => (v -> v -> v) -> [(k,v)] -> Trie k v
fromAssocAccum' f = insertAssocAccum' f empty
```

```
fromAssoc :: Ord k => [(k,v)] -> Trie k v
fromAssoc = insertAssoc empty
```

```
dumpTrie :: (Show k, Show v) => Trie k v -> String
dumpTrie (Trie p v bt)
  = "Node " ++ show p ++ " = " ++ show v ++ "\n" ++
    (unlines $ map ("  "++) $ lines kids)
  where
    kids = concatMap (\(k,t) -> show k ++ " -> " ++ dumpTrie t) (BT.assocs bt)
```

2.3 Tree

2.3.1 Introduction

A generic tree data structure. Each branch can have any number of children.

2.3.2 Data Declarations

```
data Tree a = Tree a [Tree a] deriving (Eq, Ord, Read, Show)
```

2.3.3 Labels

```
treeValue :: Tree a -> a
treeValue (Tree a _) = a

treeChildren :: Tree a -> [Tree a]
treeChildren (Tree _ kids) = kids
```

2.3.4 Folds and Traversals

```
postOrderFold :: (a -> b -> b) -> b -> Tree a -> b
postOrderFold f z (Tree me kids) = foldr (flip (postOrderFold f)) (f me z) kids

postOrder :: Tree a -> [a]
postOrder = postOrderFold (:) []
```

`postOrder` is a depth-first traversal where the root node is visited *before* its children. In Figure 2.4 the nodes would be traversed in this order: A C X Y Z D Q R E.

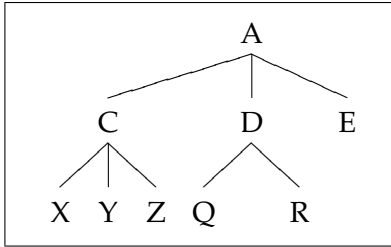


Figure 2.4: Example Tree

```

preOrderFold :: (a -> b -> b) -> b -> Tree a -> b
preOrderFold f z (Tree me kids) = f me (foldr (flip (preOrderFold f)) z kids)

preOrder :: Tree a -> [a]
preOrder = preOrderFold (:) []
  
```

`preOrder` is a depth-first traversal where the root node is visited *after* it's children. In Figure 2.4 the nodes would be traversed in this order: X Y Z C Q R D E A.

```

levelOrderFold :: (a -> b -> b) -> b -> Tree a -> b
levelOrderFold = (.levelOrder) .! foldr

levelOrder :: Tree a -> [a]
levelOrder = concat . levelOrder2

levelOrder2 :: Tree a -> [[a]]
levelOrder2 (Tree me kids) = [me] : foldr (mergeLists (++)) [] (map levelOrder2 kids)
  
```

`levelOrder` is a breadth-first traversal. All the nodes at level 0 (the root) are visited first. then all the nodes at level 1 (the root's children), etc. In Figure 2.4 the nodes would be traversed in this order: A C D E X Y Z Q R.

```

treeExtractMap :: (Tree a -> Maybe b) -> Tree a -> [b]
treeExtractMap f me@(Tree _ kids) = maybe (concatMap (treeExtractMap f) kids) (:[]) (f me)

treeExtract :: (Tree a -> Bool) -> Tree a -> [Tree a]
treeExtract = treeExtractMap . toMaybe
  
```

2.3.5 Extraction

```

treeExtractMapLevel2 :: (Tree a -> Maybe b) -> Tree a -> [[b]]
treeExtractMapLevel2 f me@(Tree _ kids) =
  maybe
    ([ : foldr (mergeLists (++)) [] (map (treeExtractMapLevel2 f) kids))
    ((:[]).(:[[]]))
    (f me)

treeExtractLevel2 :: (Tree a -> Bool) -> Tree a -> [[Tree a]]
treeExtractLevel2 = treeExtractMapLevel2 . toMaybe
  
```

```
treeExtractMapLevel :: (Tree a → Maybe b) → Tree a → [b]
treeExtractMapLevel = concat .! treeExtractMapLevel2

treeExtractLevel :: (Tree a → Bool) → Tree a → [Tree a]
treeExtractLevel = concat .! treeExtractLevel2
```

2.3.6 Mapping

```
treeMap :: (a → b) → Tree a → Tree b
treeMap f (Tree me kids) = Tree (f me) (map (treeMap f) kids)
```

2.4 Memory Buffer

2.4.1 Introduction

This is a dynamically sized memory buffer that stores `Storable`'s. It operates only in the `IO` monad. It supports fast writes and append from Foreign pointers.

2.4.2 Data Declarations

```
data Buffer a = Buffer !(IORef (ForeignPtr a)) !(IORef Int) !(IORef Int)
```

A `Buffer` consists of a `ForeignPtr` to the raw data, the size of the contents, and the capacity of the array. All these are stored in mutable `IORef`'s so the array can dynamically grow.

2.4.3 Buffer Maintenance

```
sizeOfDeref :: Storable a ⇒ Ptr a → Int
sizeOfDeref = sizeOf . (undefined::Ptr a → a)
```

`sizeOfDeref` is a helper function to find the size of an array element.

```
grow :: Storable a ⇒ Buffer a → Int → IO ()
grow (Buffer memRef sizeRef capacityRef) reqCapacity = do
  oldCapacity ← readIORef capacityRef
  let newCapacity = max reqCapacity (oldCapacity*2)
  oldMem ← readIORef memRef
  size ← readIORef sizeRef
  newMem ← mallocForeignPtrArray newCapacity
  withForeignPtr newMem $
    λnewMem' → withForeignPtr oldMem $
      λoldMem' → copyBytes newMem' oldMem' (size*sizeOfDeref newMem')
  writeIORef memRef newMem
  writeIORef capacityRef newCapacity
```

grow increases the size of an array so it can store at least reqCapacity elements.

```
addUndefined :: Storable a => Buffer a -> Int -> IO Int
addUndefined buf@(Buffer _ sizeRef capacityRef) additional = do
  size <- readIORef sizeRef
  capacity <- readIORef capacityRef
  let newSize = size + additional
  when (newSize > capacity) $ grow buf newSize
  writeIORef sizeRef newSize
  return size
```

addUndefined appends additional undefined elements to the array (growing if necessary). It returns the position of the first element added.

2.4.4 Public API

```
newBuffer :: Storable a => IO (Buffer a)
newBuffer = newBuffer' 1000

newBuffer' :: Storable a => Int -> IO (Buffer a)
newBuffer' capacity = do
  memRef <- mallocForeignPtrArray capacity >>= newIORef
  sizeRef <- newIORef 0
  capacityRef <- newIORef capacity
  return (Buffer memRef sizeRef capacityRef)
```

newBuffer creates a new Buffer. newBuffer' allows the initial capacity to be specified.

```
newBufferList :: Storable a => [a] -> IO (Buffer a)
newBufferList vs = do
  buf <- newBuffer
  bufAppendList buf vs
  return buf
```

```
bufSize :: Buffer a -> IO Int
bufSize (Buffer _ r _) = readIORef r

bufCapacity :: Buffer a -> IO Int
bufCapacity (Buffer _ _ r) = readIORef r
```

```
bufAppend :: Storable a => Buffer a -> a -> IO ()
bufAppend buf@(Buffer memRef _ _) v = do
  start <- addUndefined buf 1
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' -> pokeElemOff mem' start v
```

```
bufAppendList :: Storable a => Buffer a -> [a] -> IO ()
bufAppendList buf@(Buffer memRef _ _) vs = do
  start <- addUndefined buf (length vs)
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' ->
    sequence_ $ zipWith (pokeElemOff mem') [start..] vs
```

```

bufWrite :: Storable a => Buffer a -> Int -> a -> IO ()
bufWrite buf@(Buffer memRef sizeRef capacityRef) i v = do
  when (i < 0) $ fail "negative buffer index"
  size <- readIORef sizeRef
  capacity <- readIORef capacityRef
  when (i >= capacity) $ grow buf (i+1)
  when (i >= size) $ writeIORef sizeRef (i+1)
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' -> pokeElemOff mem' i v

```

```

bufRead :: Storable a => Buffer a -> Int -> IO a
bufRead (Buffer memRef sizeRef _) i = do
  when (i < 0) $ fail "negative buffer index"
  size <- readIORef sizeRef
  when (i >= size) $ fail "buffer index out of bounds"
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' -> peekElemOff mem' i

```

```

bufWriteRaw :: Storable a => Buffer a -> Int -> Ptr a -> Int -> IO ()
bufWriteRaw buf@(Buffer memRef sizeRef capacityRef) i ptr len = do
  when (i < 0) $ fail "negative buffer index"
  size <- readIORef sizeRef
  capacity <- readIORef capacityRef
  when (i+len > capacity) $ grow buf (i+len)
  when (i > size) $ writeIORef sizeRef (i+len)
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' ->
    copyBytes (mem' `plusPtr` (i*sizeOfDeref mem')) ptr (len*sizeOfDeref mem')

```

```

bufAppendRaw :: Storable a => Buffer a -> Ptr a -> Int -> IO ()
bufAppendRaw buf@(Buffer memRef _ _) ptr len = do
  start <- addUndefined buf len
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' ->
    copyBytes (mem' `plusPtr` (start*sizeOfDeref mem')) ptr (len*sizeOfDeref mem')

```

```

bufReadRaw :: Storable a => Buffer a -> Int -> Ptr a -> Int -> IO ()
bufReadRaw (Buffer memRef sizeRef _) i dest len = do
  when (i < 0) $ fail "negative buffer index"
  size <- readIORef sizeRef
  when (i+len > size) $ fail "buffer index out of range"
  mem <- readIORef memRef
  withForeignPtr mem $ \mem' ->
    copyBytes dest (mem' `plusPtr` (i*sizeOfDeref mem')) (len*sizeOfDeref mem')

```

```

bufElems :: Storable a => Buffer a -> IO [a]
bufElems (Buffer memRef sizeRef _) = do
  mem <- readIORef memRef
  size <- readIORef sizeRef
  withForeignPtr mem $ \mem' ->
    sequence [peekElemOff mem' i | i <- [0..size-1]]

```

Chapter 3

Data Structure Utilities

3.1 Array Functions

```
binarySearch :: (Integral i, Ix i, Ord a) => a -> Array i a -> Bool
binarySearch x a = run (bounds a)
  where
    run (s,e)
      | l < 0 = False
      | v > x = run (s,p-1)
      | v < x = run (p+1,e)
      | otherwise = True
    where
      l = e - s
      p = s + l `div` 2
      v = (a!p)
```

3.2 List Functions

```
positionPerms :: [[a]] -> [[a]]
positionPerms = foldr (\a as -> [x:xs | x <- a, xs <- as]) [[]]
```

`positionPerms` takes a list of lists of possible values for the a given position in the final output and returns all possible permutations of those values. Example:

```
positionPerms [[1,2,3],[4,5,6]] = [
  [1,4], [1,5], [1,6], [2,4], [2,5], [2,6], [3,4], [3,5], [3,6]
]
```

```
listPerms :: Eq a => Int -> [a] -> [[a]]
listPerms 0 _ = [[]]
listPerms _ [] = []
listPerms n xs = [x:ps | x<-xs, ps <- listPerms (n-1) (delete x xs)]
```

```
allListPerms :: Eq a => [a] -> [[a]]
allListPerms xs = concatMap (flip listPerms xs) $ map succ $ listIndices xs
```

`listPerms` takes a list of elements and returns all the different n element permutations of those elements.

```
listIndices :: [a] -> [Int]
listIndices = listIndicesFold (:) []

listIndicesFold :: (Int -> b -> b) -> b -> [a] -> b
listIndicesFold c n = f 0
  where f _ [] = n
        f n (_:xs) = n `c` f (n+1) xs
```

`listIndices` maps the elements of a list to their index in the list. The head of this list is always 0 and last element is always the list's length minus 1.

```
xs = listIndices ["Foo", "Bar", "Baz"] == [0,1,2]
```

```
#ifdef __GLASGOW_HASKELL__
{-# RULES
"listIndices" [~1] forall xs. listIndices xs = build (\c n -> listIndicesFold c n xs)
"listIndicesFold/build"
  forall c n (g::forall b.(a->b)->b) .
  listIndicesFold c n (build g) = g (\_ r v -> v `c` r (v+1)) (\_ -> n) 0
-#}
#endif
```

These are deforestation rules for GHC. `listIndices` is both a good producer and consumer.

```
chunkifyExact :: Int -> [a] -> [[a]]
chunkifyExact n w
  | null e = if length s == n then [s] else []
  | otherwise = s : chunkifyExact n e
  where (s,e) = splitAt n w

chunkify :: Int -> [a] -> [[a]]
chunkify _ [] = []
chunkify n w = s : chunkify n e
  where (s,e) = splitAt n w
```

`chunkify` and `chunkifyExact` take a list and split it up into smaller lists of size n . If there isn't enough input to evenly fit into a chunk the `chunkifyExact` discards the extra data.

```

pad :: a -> Int -> [a] -> [a]
pad = padFold (:) []

padFold :: (a -> b -> b) -> b -> a -> Int -> [a] -> b
padFold c n p x = pad' 0
  where
    pad' i [] = padEnd c n p x i
    pad' i (x:xs) = x 'c' pad' (i+1) xs

padEnd :: (a -> b -> b) -> b -> a -> Int -> Int -> b
padEnd c n p x i
  | r == 0 = n
  | otherwise = foldr c n $ take (x - r) $ repeat p
  where r = i `rem` x

```

pad pads a list so it is a multiple of the given size.

```

#ifdef __GLASGOW_HASKELL__
{-# RULES
"pad" [~1] forall p x xs. pad p x xs = build (\c n -> padFold c n p x xs)
"padFold/build"
forall c n p x (g::forall b.(a->b->b->b) .
padFold c n p x (build g) = g (\x r i -> x 'c' r (i+1)) (padEnd c n p x) 0
#-}
#endif

```

These are deforestation rules for GHC. pad is both a good producer and consumer.

```

takeEvery :: Int -> [a] -> [a]
takeEvery = takeEveryFold (:) []

takeEveryFold :: (a -> b -> b) -> b -> Int -> [a] -> b
takeEveryFold _ n _ [] = n
takeEveryFold c n i (x:xs) = x 'c' (takeEveryFold c n i $ drop (i-1) xs)

```

takeEvery returns a list containing every *n*th element of the given list. Example:

```
takeEvery 3 [1..10] = [1,4,7,10]
```

```

#ifdef __GLASGOW_HASKELL__
{-# RULES
"takeEvery" [~1] forall i xs. takeEvery i xs = build(\c n -> takeEveryFold c n i xs)
"takeEveryFold/build"
forall c n i (g::forall b.(a->b->b->b) .
takeEveryFold c n i (build g) = g (\x r s ->
if s==0
then x 'c' r (i-1)
else r (s-1)) (\_ -> n) 0
#-}
#endif

```

These are deforestation rules for GHC. takeEvery is both a good producer and consumer.

```

keySortBy :: (a -> a -> Ordering) -> [(a,b)] -> [b]
keySortBy c = map snd . sortBy (\(a,_) (b,_) -> c a b)

keySort :: Ord a => [(a,b)] -> [b]
keySort = keySortBy compare

keySortRev :: Ord a => [(a,b)] -> [b]
keySortRev = keySortBy (flip compare)

```

keySortBy sorts a list of key/value pairs by the key and returns the sorted values.

```

sortedBy' :: (a -> a -> Bool) -> [a] -> Bool
sortedBy' before (x:xs) = f x xs
  where
    f y (x:xs) = if y `before` x then f x xs else False
    f _ _ = True
sortedBy' _ [] = True

sortedBy :: (a -> a -> Ordering) -> [a] -> Bool
sortedBy c = sortedBy' (\x y -> x `c` y /= GT)

sorted :: Ord a => [a] -> Bool
sorted = sortedBy compare

```

```

splitOn :: (a -> Bool) -> [a] -> ([a],[a])
splitOn _ [] = ([],[a])
splitOn f (x:xs)
  | f x = ([],[a])
  | otherwise = (x:xs',ys)
  where (xs',ys) = splitOn f xs

splitManyOn :: (a -> Bool) -> [a] -> [[a]]
splitManyOn f xs =
  case splitOn f xs of
    (xs',[]) -> [xs']
    (xs',ys) -> xs':splitManyOn f ys

```

splitOn and splitOnMany split a list up into pieces. splitOn is different from the prelude function splitAt in that the matched item is not included in the second list.

```

leftTrim :: (a -> Bool) -> [a] -> [a]
leftTrim = dropWhile

rightTrim :: (a -> Bool) -> [a] -> [a]
rightTrim f xs = case break f xs of
  (good,xs) -> case break (not.f) xs of
    (_,[]) -> good
    (spaces,rest) -> good ++ spaces ++ rightTrim f rest

trim :: (a -> Bool) -> [a] -> [a]
trim f = rightTrim f . leftTrim f

startsWith :: Eq a => [a] -> [a] -> Bool
startsWith _ [] = True
startsWith [] _ = False
startsWith (x:xs) (y:ys) = x == y && startsWith xs ys

endsWith :: Eq a => [a] -> [a] -> Bool
endsWith x y = startsWith (reverse x) (reverse y)

```

ltrim, rtrim, and trim remove leading, trailing, or leading and trailing elements from a list that match the given function.

```
skip :: Int -> [a] -> [a]
skip _ [] = error "Brianweb.skip: not enough list elements for skip"
skip 0 (_:xs) = xs
skip i (x:xs) = x : skip (i-1) xs

skipFold :: (a -> b -> b) -> b -> Int -> [a] -> b
skipFold _ _ _ [] = error "Brianweb.skip: not enough list elements for skip"
skipFold c n 0 (_:xs) = foldr c n xs
skipFold c n i (x:xs) = x `c` skipFold c n (i-1) xs
```

skip skips element p of the given list

```
#ifdef __GLASGOW_HASKELL__
{-# RULES
"skip" [~1] forall i xs. skip i xs = build (\c n -> skipFold c n i xs)
"skipFold/build"
forall c n i (g::forall b.(a->b->b->b) .
skipFold c n i (build g) = g (\x r i -> (if i==0 then id else (x`c`)) (r (i-1))) (\_ -> n) i
— FIXME: This isn't firing
"skipList" [1] forall i xs. skipFold (:) [] i xs = skip i xs
#-}
#endif
```

These are deforestation rules for GHC. skip is both a good producer and consumer.

```
sortedElem :: Ord a => a -> [a] -> Bool
sortedElem _ [] = False
sortedElem x' (x:xs) =
  case compare x x' of
    EQ -> True
    GT -> False
    LT -> sortedElem x' xs
```

sortedElem returns true if an element exists in a sorted list. The search stops once a list element is found greater than the element being searched for. This means sortedElem can be used on infinite lists.

```
uniq :: Eq a => [a] -> [a]
uniq = uniqFold (:) []

uniqFold :: Eq a => (a -> b -> b) -> b -> [a] -> b
uniqFold _ n [] = n
uniqFold c n (x:xs) = x `c` uniqFold c n (dropWhile (==x) xs)
```

uniq works just like the UNIX command uniq. It is equivalent to map head . group, but faster.

```
#ifdef __GLASGOW_HASKELL__
{-# RULES
"uniq" [~1] forall xs. uniq xs = build (\c n -> uniqFold c n xs)
#-}
#endif
```

These are deforestation rules for GHC. `uniq` is a good producer.

```
xorList :: [Bool] → Bool
xorList = foldr (/=) False
```

```
sortByField :: Ord key ⇒ (a → key) → [a] → [a]
sortByField f = sortBy (λa b → compare (f a) (f b))

groupByField :: Eq key ⇒ (a → key) → [a] → [[a]]
groupByField f = groupBy (λa b → f a == f b)

groupByWhile :: (a → a → Bool) → [a] → [[a]]
groupByWhile _ [] = []
groupByWhile f (x:xs) = (x:ys) : groupWhile f zs
  where (ys,zs) = spanWhile f x xs

spanWhile :: (a → a → Bool) → a → [a] → ([a],[a])
spanWhile _ _ [] = ([],[a])
spanWhile f prev xs@(x:xs)
  | f prev x = (x:ys,zs)
  | otherwise = ([],xs')
  where (ys,zs) = spanWhile f x xs
```

```
interleave :: [a] → [a] → [a]
interleave [] ys = ys
interleave (x:xs) ys = x : (ys `interleave` xs)

powerSet :: [a] → [[a]]
powerSet [] = [[]]
powerSet (x:xs) = ys `interleave` map (x:) ys
  where ys = powerSet xs

subSet :: Int → [a] → [[a]]
subSet 0 _ = [[]]
subSet _ [] = []
subSet n (x:xs)
  | n > 0 = map (x:) (subSet (n-1) xs) `interleave` subSet n xs
  | otherwise = error "Brianweb.Data.List.subSet: negative n"
```

```
mergeLists :: (a → a → a) → [a] → [a] → [a]
mergeLists f (x:xs) (y:ys) = f x y : mergeLists f xs ys
mergeLists _ xs [] = xs
mergeLists _ [] ys = ys
```

```
unfoldr :: (b → (Maybe a,b)) → b → ([a],b)
unfoldr f b = case f b of
  (Just a,b') → let (as,b) = unfoldr f b' in (a:as,b)
  (Nothing,b') → ([],b')
```

```
unfoldl :: (a → (Maybe b,a)) → a → ([b],a)
unfoldl f a = go a []
  where
    go a bs = case f a of
      (Just b,a') → go a' (b:bs)
      (Nothing,a') → (bs,a')
```

```
swap :: Eq a => a -> a -> [a] -> [a]
swap a b (x:xs)
  | x == a = b : swap a b xs
  | x == b = a : swap a b xs
  | otherwise = x : swap a b xs
swap _ _ [] = []
```

```
notNull :: [a] -> Bool
notNull [] = False
notNull (_:_) = True
```

```
rdlof :: [a] -> b -> (a -> b -> b) -> b
rdlof xs z f = foldr f z xs
```

```
agree :: Eq b => (a -> b) -> [a] -> Maybe b
agree _ [] = Just (error "Brianweb.Data.List.agree: empty list")
agree f (x:xs) = foldl g (Just (f x)) xs
  where
    g (Just x) y | x == f y = Just x
    g _ _ = Nothing
```

`agree` tests if every element of a given list agree on the result of some test. If so, `Just r` is returned, where `r` is the result. If not, `Nothing` is returned. The empty list returns `Just bottom`

3.3 Maybe Functions

```
toMaybe :: (a -> Bool) -> a -> Maybe a
toMaybe f x
  | f x = Just x
  | otherwise = Nothing
```

```
firstJust :: [Maybe a] -> Maybe a
firstJust = msum
```

Chapter 4

Monads

4.1 Failure Monad

A failure monad is any monad that can express failure in some sane way `fail` in the monad class doesn't count. It is often mapped to `error` and doesn't give the user any indication that the computation may fail. `fail` shouldn't be part of the `Monad` class. `MonadFail` is simply the failure part `MonadError` (since not all monads can recover from failure).

```
class Monad m => MonadFail m where
  mfail :: String -> m a
```

A `MonadFail` instance provides a `mfail` function which expresses failure within the monad. The description argument may be ignored by some monads.

```
instance MonadFail Identity where
  mfail = error

instance MonadFail Maybe where
  mfail _ = Nothing

instance MonadFail IO where
  mfail = ioError . userError
```

`MonadFail` instances for some common monads.

```
instance MonadFail m => MonadFail (ReaderT r m) where
  mfail = lift . mfail

instance MonadFail m => MonadFail (StateT s m) where
  mfail = lift . mfail
```

`MonadFail` instances for some common monad transformers.

```

newtype Result a = Result { runResult :: Either String a }

instance Monad Result where
  return = Result . Right
  (Result (Right a)) >>= f = f a
  (Result (Left s)) >>= _ = Result (Left s)

```

The result monad is just like the Maybe monad but with an error string.

```

instance MonadFail Result where
  mfail = Result . Left

instance MonadFix Result where
  mfix f = let a = f (unRight a) in a
    where
      unRight (Result (Right x)) = x
      unRight _ = undefined

```

MonadFail and MonadFix instances for the result monad.

4.2 Misc Monad Functions

```

apl :: (Monad m, Monad (t m), MonadTrans t) => t m (a -> b) -> m a -> t m b
apl f x = f `ap` lift x

```

apl is simply ap with its first argument lifted.

```

forEachM :: Monad m => [a] -> (a -> m b) -> m [b]
forEachM = flip mapM

forEachM_ :: Monad m => [a] -> (a -> m b) -> m ()
forEachM_ = flip mapM_

```

forEachM is a flipped version for mapM.

Chapter 5

Streams

5.1 Input Streams

```
newtype Stream m a = S { read :: m (Maybe a) }  
  
type StreamTrans m a b = Stream m a → Stream m b  
  
instance Monad m ⇒ Functor (Stream m) where  
    fmap = map
```

```
mkStream :: m (Maybe a) → Stream m a  
mkStream = S  
  
map :: Monad m ⇒ (a → b) → Stream m a → Stream m b  
map f s = S (liftM (fmap f) (read s))
```

```
read' :: MonadFail m ⇒ Stream m a → m a  
read' is = read is >>= maybe (mfail "eof unexpected") return
```

```
handle :: Handle → Stream IO Char  
handle h = S $ do  
    eof ← hIsEOF h  
    if eof  
        then return Nothing  
        else liftM Just (hGetChar h)  
  
withFile :: (Stream IO Char → IO a) → FilePath → IO a  
withFile f fp = bracket (openFile fp ReadMode) hClose (f . handle)
```

```
list :: Monad m => Stream (StateT [a] m) a
list = S $ do
  xs' ← get
  case xs' of
    [] → return Nothing
    (x:xs) → put xs >> return (Just x)

runList :: Monad m => StateT [a] m b → [a] → m b
runList = evalStateT
```

```

makeBinary :: Monad m => StreamTrans m Char Word8
makeBinary = map (fromIntegral . ord) — FEATURE: This isn't safe, need a real binary file reader

transform :: Monad m => (a -> m b) -> StreamTrans m a b
transform f is = S $ read is >>= maybe (return Nothing) (\x -> f x >>= return.Just)

utf8Reader :: MonadFail m => StreamTrans m Word8 Char
utf8Reader is = transform f is
  where
    f b
      | c <= 0x7f = do — 0xxxxxxx
        return $ chr c
      | c <= 0xdf = do — 110xxxxx 10xxxxxx
        c2 ← next
        return $ chr $ (c.&.0x1f)'shiftL' 6 + c2
      | c <= 0xef = do — 1110xxxx 10xxxxxx 10xxxxxx
        c2 ← next; c3 ← next
        return $ chr $ (c.&.0x0f)'shiftL'12 + c2'shiftL'6 + c3
      | c <= 0xf4 = do — 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx (but limited to 0x10ffff)
        c2 ← next; c3 ← next; c4 ← next
        return $ chr $ (c.&.0x07)'shiftL'18 + c2'shiftL'12 + c3'shiftL'6 + c4
      | otherwise = mfail "invalid UTF-8 encoding"
    where
      next = read' is >>= return.(.&.0x3f).fromIntegral
      c = fromIntegral b

utf16Reader :: MonadFail m => StreamTrans m Word16 Char
utf16Reader is = transform f is
  where
    f c
      | c >= 0xd800 && c <= 0xdbff = do — 110110xxxxxxxxxxxx 110111xxxxxxxxxxxx
        c2 ← read' is
        return $ chr $ 0x10000 + (fromIntegral c .&. 0x3ff)'shiftL'10 +
          (fromIntegral c2 .&. 0x3ff)
      | otherwise = return $ chr $ fromIntegral c

ceus8Reader :: MonadFail m => StreamTrans m Word8 Char
ceus8Reader = utf16Reader . transform (f . fromIntegral . ord) . utf8Reader
  where
    f x
      | x <= 0xffff = return x
      | otherwise = mfail "invalid CEUS-8 encoding"

modifiedUtf8Reader :: MonadFail m => StreamTrans m Word8 Char
modifiedUtf8Reader = ceus8Reader

getContentsLazy :: Monad m => Stream m a -> m [a]
getContentsLazy s = f
  where
    f = read s >>= maybe (return []) (\x -> liftM (x:) f)

getContentsStrict :: Monad m => Stream m a -> m [a]
getContentsStrict s = f id
  where
    f c = read s >>= maybe (return (c [])) (\x -> f (c . (x:)))

```

```

counted :: Monad m => Stream m a -> Stream (StateT Int m) a
counted s = S $ do
  e <- lift (read s)
  when (isJust e) $ modify (+1)
  return e

runCounted :: Monad m => StateT Int m a -> m a
runCounted = (`evalStateT`0)

limited :: Monad m => Stream m a -> Int -> Stream (StateT Int m) a
limited s limit = S $ do
  p <- get
  if p < limit
    then read (counted s)
    else return Nothing

```

5.2 Output Streams

```

newtype Stream m a = S { write :: a -> m () }

type StreamTrans m a b = Stream m a -> Stream m b

```

```

mkStream :: (a -> m ()) -> Stream m a
mkStream = S

map :: (b -> a) -> Stream m a -> Stream m b
map f s = S (write s . f)

```

```

handle :: Handle -> Stream IO Char
handle = S . hPutChar

withFile :: (Stream IO Char -> IO a) -> FilePath -> IO a
withFile f fp = bracket (openFile fp WriteMode) hClose (f . handle)

```

```

lazyList :: Monad m => Stream (ContT (r,[a]) m) a
lazyList = S $ \x -> ContT $ \c -> do
  ~(r,xs) <- c ()
  return (r,x:xs)

runLazyList :: Monad m => ContT (r,[a]) m r -> m (r,[a])
runLazyList m = runContT m (\r -> return (r,[]))

```

```

strictList :: Monad m => Stream (StateT ([a]->[a]) m) a
strictList = S $ \x -> modify (.(x:))

runStrictList :: Monad m => StateT ([a]->[a]) m r -> m (r,[a])
runStrictList m = do
  (x,c) <- runStateT m id
  return (x,c [])

```

```

null :: Monad m => Stream m a
null = S (\_ -> return ())

```

```

makeBinary :: Monad m => StreamTrans m Char Word8
makeBinary = map (chr . fromIntegral) — FEATURE: This isn't safe, need a real binary file writer

writeString :: Monad m => Stream m e -> [e] -> m ()
writeString = mapM_ . write

```

```

utf8Writer :: Monad m => StreamTrans m Word8 Char
utf8Writer s = mkStream f
  where
    f c'
      | c <= 0x7f = do
          w c
      | c <= 0x7ff = do
          w (c `shiftR` 6 .|. 0xc0)
          w (c .&. 0x3f .|. 0x80)
      | c <= 0xffff = do
          w (c `shiftR` 12 .|. 0xe0)
          w (c `shiftR` 6 .&. 0x3f .|. 0x80)
          w (c .&. 0x3f .|. 0x80)
      | c <= 0x10ffff = do
          w (c `shiftR` 18 .|. 0xf0)
          w (c `shiftR` 12 .&. 0x3f .|. 0x80)
          w (c `shiftR` 6 .&. 0x3f .|. 0x80)
          w (c .&. 0x3f .|. 0x80)
      | otherwise = error "Char outside unicode range"
    where
      w = write s . fromIntegral
      c = ord c'

utf16Writer :: Monad m => StreamTrans m Word16 Char
utf16Writer s = mkStream f
  where
    f c'
      | c <= 0xffff = w c
      | c <= 0x10ffff = do
          let x = c - 0x10000
              w (x `shiftR` 10 .|. 0xd800)
              w (x .&. 0x3ff .|. 0xdc00)
          | otherwise = error "Char outside unicode range"
    where
      w = write s . fromIntegral
      c = ord c'

ceus8Writer :: Monad m => StreamTrans m Word8 Char
ceus8Writer = utf16Writer . map (chr . fromIntegral) . utf8Writer

modifiedUtf8Writer :: Monad m => StreamTrans m Word8 Char
modifiedUtf8Writer s = mkStream f
  where
    f '\NUL' = write s 0xc0 >> write s 0x80
    f c = write (ceus8Writer s) c

```

```

counted :: Monad m => Stream m a -> Stream (StateT Int m) a
counted is = S (\x -> lift (write is x) >> modify (+1))

runCounted :: Monad m => StateT Int m a -> m a
runCounted = ('evalStateT' 0)

```

```

delayed :: (Monad m, Monad m') => Stream m' e -> Stream (StateT (m' ()) m) e
delayed is = mkStream (modify . flip (>>) . write is)

runDelayed :: (Monad m, Monad m') => StateT (m' ()) m a -> m (a, (m' ()))
runDelayed = (`runStateT`return ())

```

5.3 Consumers

```

type Consumer m e a = ReaderT (I.Stream m e) m a

runConsumer :: Monad m => Consumer m e a -> I.Stream m e -> m a
runConsumer = runReaderT

consume' :: Monad m => Consumer m a (Maybe a)
consume' = ask >>= lift . I.read

consume :: MonadFail m => Consumer m a a
consume = consume' >>= maybe (mfail "unexpected eof") return

consumeEOF :: MonadFail m => Consumer m a ()
consumeEOF = consume' >>= maybe (return ()) (\_ -> mfail "eof expected")

```

5.4 Producers

```

type Producer m e a = ReaderT (O.Stream m e) m a

runProducer :: Monad m => Producer m e a -> O.Stream m e -> m a
runProducer = runReaderT

produce :: Monad m => e -> Producer m e ()
produce x = ask >>= \s -> lift (O.write s x)

```

Chapter 6

Text Processing and Parsers

6.1 CSV Library

6.1.1 Introduction

This module contains functions for parsing and writing CSV files. It can properly read and write CSV files with fields containing any character (including newlines), however, many other applications cannot deal with things like this. Care should be taken when using any kind of special characters in CSV fields.

6.1.2 Parsing

```
parseCSV :: Parser ([String],[String])
parseCSV = do
  headers ← parseCSVLine
  rest ← many (checkedLine (length headers))
  return (headers,rest)
  where
    checkedLine n = do
      fs ← parseCSVLine
      when (length fs ≠ n) (fail "Not enough fields in line")
      return fs
```

```
parseHeaderlessCSV :: Parser [[String]]
parseHeaderlessCSV = many parseCSVLine
```

```
parseCSVFieldSep :: Parser ()
parseCSVFieldSep = between s s (do { char ','; return (); })
  where s = many (oneOf " \t")
```

```

parseCSVLine :: Parser [String]
parseCSVLine = do
  fs ← parseCSVField `sepBy` parseCSVFieldSep
  do { char '\r'; char '\n' } <|> char '\n'
  return fs

```

```

parseCSVField :: Parser String
parseCSVField = parseCSVEnclosedField <|> parseCSVRawField

```

```

parseCSVEnclosedField :: Parser String
parseCSVEnclosedField = do
  char '"'
  s ← many (noneOf "\"" <|> try(do { char '"'; char '"'; }))
  char '"'
  return s

```

```

parseCSVRawField :: Parser String
parseCSVRawField = many (noneOf "\r\n")

```

```

readCSV :: FilePath → IO ([String],[[String]])
readCSV f = parseFromFile parseCSV f >>= fromEitherM

```

6.1.3 Writing

```

encodeField :: String → String
encodeField = ('":) . f
  where
    f [] = ""
    f (x:xs)
      | x == '"' = ':' : f xs
      | otherwise = x : f xs

```

```

— FIXME: This is slow (or is it? deforestation?)
writeCSV' :: ([String],[[String]]) → String
writeCSV' (fs,rs) = unlines (fl:rls)
  where
    fl = concat $ intersperse "," (map encodeField fs)
    rls = map (concat . intersperse "," . map encodeField) rs

writeCSV :: FilePath → ([String],[[String]]) → IO()
writeCSV f csv = writeFile f (writeCSV' csv)

```

6.1.4 Map Conversion

```
csvToBTList :: ([String],[[String]]) → [BT.Map String String]
csvToBTList (header,records) = map (BT.fromAssocs . zip header) records

btListToCSV :: [BT.Map String String] → ([String],[[String]])
btListToCSV bts = (keys,values)
  where
    keys = BT.keys $ foldr BT.union BT.empty bts
    values = map (λrow → map (fromMaybe " " . flip BT.lookup row) keys) bts
```

6.2 Parsec Functions

There are simple convenience functions for writing parsers.

```
maybeParse :: GenParser tok st a → GenParser tok st (Maybe a)
maybeParse p = option Nothing (p >>= return.Just)
```

```
notFollowedBy' :: Show a ⇒ GenParser tok st a → GenParser tok st ()
notFollowedBy' p = try (do { c ← p; unexpected (show c) } <|> return ())
```

```
parseUpTo :: Int → GenParser tok st a → GenParser tok st [a]
parseUpTo n p
  | n ≤ 0 = return []
  | otherwise = do {
    i ← p;
    rest ← parseUpTo (n-1) p;
    return (i:rest);
  } <|> return []
```

```
followedBy :: GenParser tok st end → GenParser tok st a → GenParser tok st a
followedBy end p = p >>= λx → end >> return x
```

```
withState :: st → GenParser tok st a → GenParser tok st a
withState s = withStateChange (λ_ → s)
```

```
withStateChange :: (st → st) → GenParser tok st a → GenParser tok st a
withStateChange f p = do
  oldState ← getState
  setState (f oldState)
  ret ← p
  setState oldState
  return ret
```

```

parserToReadS :: Parser a → ReadS a
parserToReadS p s = either (const []) (:[]) (parse p' "" s)
  where
    p' = do
      a ← p
      rest ← getInput
      return (a, rest)

```

6.3 HTML Parser

6.3.1 Introduction

This module is the opposite of `Text.Html`. It parses an HTML document into an `Html` structure. Unlike most HTML/XML parsers this one is *very* forgiving. It'll handle tags that aren't closed properly, attributes not enclosed in quotes, improperly nested block elements, random characters between attributes, etc. The goal is to be able to parse any HTML document found "in the wild" however, it certainly isn't there yet. It will still fail on some inputs.

```

type Parser a = GenParser Char [String] a

```

6.3.2 Stack management

We keep track of what tags are currently open while parsing the document. When an unexpected close tag is encountered it'll be ignored if it doesn't match an already open tag.

```

pushTag :: String → Parser ()
pushTag = updateState . (:)

popTag :: Parser ()
popTag = updateState tail

tagInStack :: String → Parser Bool
tagInStack t = getState >>= return . (t `elem`)

```

6.3.3 Public Interface

```
parseHtmlDoc :: String → String → Either ParseError Html
parseHtmlDoc fn s = case parsed of
  Right (Html xs) → case filter (
    λx → case x of HtmlString s → not $ all isSpace s; _ → True
  ) xs of
    [HtmlTag "html" _ content] → Right content
    _ → Right $ Html xs
  _ → parsed
where
  parsed = runParser p [] fn s
  p = do
    html ← parseHtml
    eof
    return html
```

6.3.4 Parser

```
parseHtml :: Parser Html
parseHtml = parseHtmlElements >>= return.Html

parseHtmlElements :: Parser [HtmlElement]
parseHtmlElements = many parseHtmlElement >>= return.concat
```

```
parseHtmlElement :: Parser [HtmlElement]
parseHtmlElement = parseTagOrComment <|> parseHtmlString
```

```
parseHtmlString :: Parser [HtmlElement]
parseHtmlString = many1 (satisfy (≠'<')) >>= return.(:[]).HtmlString
```

```
parseTagOrComment :: Parser [HtmlElement]
parseTagOrComment = do
  done ← try (char '<' >> (checkCloseTag <|> return False))
  if done
    then return []
    else continueTag <|> (continueJunkTag >> return [])
  where
    checkCloseTag = do
      y ← continueCloseTag >>= tagInStack
      if y
        then pzero — Let someone else eat the tag
        else return True — Don't keep going, we ate it
```

```
continueJunkTag :: Parser ()
continueJunkTag = do
  c ← oneOf "!"
  when (c=='!') continueComment
  finishTag
```

```
finishTag :: Parser ()
finishTag = manyTill anyChar (char '>') >> return ()
```

```

continueComment :: Parser ()
continueComment = sepBy spaces singleComment >> return ()
  where
    singleComment = try (string "--") >> manyTill anyChar (try $ string "--") >> return ()

```

```

continueTag :: Parser [HtmlElement]
continueTag = do
  tag ← parseName
  spaces
  attrs ← parseHtmlAttr `sepEndBy` (many space)
  closed ← maybeParse (try $ char '/' >> spaces) >>= return.isJust
  char '>'
  case () of
    () | tag `elem` ["xmp","script"] → do
      content ← manyTill anyChar $ try $ parseCloseTag >>= λtag' → guard (tag'==tag)
      return [HtmlTag tag attrs (Html [HtmlString content])]
    () | closed || emptyTag tag → do
      when (not closed) $ optional $ try $ parseCloseTag >>= λtag' → guard (tag==tag')
      return [HtmlTag tag attrs (Html [])]
    () → do
      pushTag tag
      content ← parseHtmlElements
      popTag
      optional $ try $ parseCloseTag >>= λtag' → guard (tag==tag')
      return $ saneHtmlTag tag attrs content

```

```

parseCloseTag :: Parser String
parseCloseTag = char '<' >> continueCloseTag

continueCloseTag :: Parser String
continueCloseTag = do
  char '/'
  tag ← parseName
  spaces
  char '>'
  return tag

```

```

parseHtmlAttr :: Parser HtmlAttr
parseHtmlAttr = do
  name ← parseName
  value ← (try (spaces >> char '=') >> spaces >> parseValue) <|> return name
  optional (oneOf ";,") — HACK for really broken html
  return $ HtmlAttr name value
  where
    parseValue = delimValue <|> rawValue
    delimValue = do
      c ← oneOf "'\λ"
      s ← manyTill anyChar (char c <|> char '>')
      return $ if c == 'λ'
        then (concatMap fixQuote s)
        else s
    rawValue = many (alphaNum <|> oneOf ".-#()/%_?=:")
    fixQuote '"' = "&quot;"
    fixQuote c = [c]

```

```

parseName :: Parser String
parseName = do
  c ← letter
  cs ← many (alphaNum <|> oneOf "-:")
  return $ map toLower (c:cs)

```

6.3.5 HTML Cleanup

These functions deal with cleaning up all sorts of messiness in the HTML document. They check for (and fix) improperly nested block level tags and help the parser decide how to parse certain tags.

```

emptyTag :: String → Bool
emptyTag = (\`elem` ["img","br","link","area","param","hr","input","base","meta","option"])

```

```

saneHtmlTag :: String → [HtmlAttr] → [HtmlElement] → [HtmlElement]
saneHtmlTag tag attrs children
  | allowBlockLevelChildren tag = [HtmlTag tag attrs (Html children)]
  | otherwise = HtmlTag tag attrs (Html inside) : outside
  where
    (inside,outside) = break blockLevelElement children

```

```

blockLevelElement :: HtmlElement → Bool
blockLevelElement (HtmlTag tag _ _) = blockLevelTag tag
blockLevelElement _ = False

```

```

blockLevelTag :: String → Bool
blockLevelTag ['h',n]
  | n `elem` ['1','2','3','4','5','6'] = True
blockLevelTag t
  | t `elem` ["p","pre","blockquote","div","address"] = True
  | otherwise = False

```

```

— FIXME: This is probably incomplete
allowBlockLevelChildren :: String → Bool
allowBlockLevelChildren t
  | t `elem` ["div","td","blockquote","body"] = True
  | otherwise = False

```

6.4 Html Utilities

6.4.1 Introduction

This module contains functions designed to be used with the data structures in `Text.Html`. It provides functions to easily extract data from the `Html` and `HtmlElement` data structures. It is primarily designed to be used for things like screen scraping in conjunction with `Html.Parser`.

6.4.2 Data Structures

```
data Form = Form {
  formAction :: String,
  formMethod :: String,
  formItems  :: [FormItem]
} deriving (Eq,Show)

data FormItem = FormItem {
  formItemName :: String,
  formItemValue :: String,
  formItemValues :: [String]
} deriving (Eq,Show)
```

6.4.3 Instance Declarations

```
instance Eq HtmlAttr where
  (HtmlAttr k1 v1) == (HtmlAttr k2 v2) = k1 == k2 && v1 == v2

instance Ord HtmlAttr where
  compare (HtmlAttr k1 v1) (HtmlAttr k2 v2) = compare (k1,v1) (k2,v2)
```

```
instance Eq Html where
  (Html x) == (Html y) = mergeHtmlStrings x == mergeHtmlStrings y

instance Eq HtmlElement where
  (HtmlString x) == (HtmlString y) = words x == words y
  (HtmlTag n1 a1 c1) == (HtmlTag n2 a2 c2) = n1 == n2 && sort a1 == sort a2 && c1 == c2
  _ == _ = False
```

```
instance Show HtmlElement where
  showsPrec _ = renderHtml' 0
```

```
mergeHtmlStrings :: [HtmlElement] → [HtmlElement]
mergeHtmlStrings ((HtmlString s1):(HtmlString s2):xs) = mergeHtmlStrings (HtmlString (s1++s2):xs)
mergeHtmlStrings (x:xs) = x : mergeHtmlStrings xs
mergeHtmlStrings [] = []
```

6.4.4 Tree Conversion

Brianweb.Data.Tree contains many useful functions for operating on generic Tree data structures. These functions convert Html and HtmlElement to Tree HtmlElement.

```
htmlToTree :: Html → Tree HtmlElement
htmlToTree = htmlElToTree . HtmlTag "" []

htmlElToTree :: HtmlElement → Tree HtmlElement
htmlElToTree e@(HtmlString _) = Tree e []
htmlElToTree e@(HtmlTag _ _ (Html kids)) = Tree e (map htmlElToTree kids)
```

6.4.5 Misc Function

```
lookupAttr :: String → [HtmlAttr] → Maybe String
lookupAttr k = lookup k . map (λ(HtmlAttr k v) → (k,v))
```

6.4.6 Text Conversion

The following functions are used to turn HTML into plain text while still maintaining some formatting (using **Bold**, */Italic/*, etc).

```
textOnlyHtmlShows :: Html → Shows
textOnlyHtmlShows (Html xs) z = foldr textOnlyElement z xs
  where
    textOnlyElement (HtmlString s) = (unescapeHtml s++)
    textOnlyElement (HtmlTag tag _ kids)
      | tag == "b" || tag == "strong" = ('*':) . kids' . ('*':)
      | tag == "i" || tag == "em" = ('/':) . kids' . ('/':)
      | tag == "u" = ('_':) . kids' . ('_':)
      | tag == "img" = ("[Image]"++)
      | otherwise = kids'
    where
      kids' = textOnlyHtmlShows kids
```

```
textOnlyHtml :: Html → String
textOnlyHtml = flip textOnlyHtmlShows []

textOnlyHtmlElement :: HtmlElement → String
textOnlyHtmlElement = textOnlyHtml . Html . (:[])
```

```
unescapeHtml :: String → String
unescapeHtml ('&':xs) = match escapes
  where
    escapes = [("nbsp", " "), ("lt", "<"), ("gt", ">"), ("quot", "λ")]
    match [] = '&' : unescapeHtml xs
    match ((e,v):es)
      | h == (e++";") = v ++ unescapeHtml t
      | otherwise = match es
      where (h,t) = splitAt (length e + 1) xs
    unescapeHtml (x:xs) = x : unescapeHtml xs
    unescapeHtml [] = ""
```

6.4.7 Data Extraction Functions

These function try to extract various pieces of information from an Html document and turn them into a more useful data structure.

```

extractLinks :: Html → [(Html, String)]
extractLinks = treeExtractMap (f.treeValue) . htmlToTree
  where
    f (HtmlTag "a" attrs content) = fmap (,) content ("href" `lookupAttr` attrs)
    f _ = Nothing

extractTextLinks :: Html → [(String, String)]
extractTextLinks = map (λ(h,l) → (textOnlyHtml h,l)) . extractLinks

```

```

extractTable :: HtmlElement → Maybe [[Html]]
extractTable (HtmlTag "table" _ kids) = Just $ treeExtractMap row $ htmlToTree kids
  where
    row me@(Tree (HtmlTag "tr" _ _) _) = Just (treeExtractMap col me)
    row _ = Nothing
    col (Tree (HtmlTag t _ html) _) | t=="th"||t=="td" = Just html
    col _ = Nothing
extractTable _ = Nothing

```

```

extractFromPage :: (HtmlElement → Maybe a) → Html → Maybe a
extractFromPage p = listToMaybe . treeExtractMapLevel (p.treeValue) . htmlToTree

```

`extractFromPage` searches through the HTML tree top down looking for an element that satisfies the predicate. This is useful in conjunction with the other extract functions.

```

extractTitle :: Html → Maybe String
extractTitle
  = fmap (textOnlyHtml . markupContent . treeValue)
  . listToMaybe . treeExtract p . htmlToTree
  where
    p (Tree (HtmlTag "title" _ _) _) = True
    p _ = False

```

```

findTable :: ([String] → Bool) → Html → Maybe [[Html]]
findTable p = extractFromPage f
  where
    f e = do
      (h:t) ← extractTable e
      guard $ p $ map (trim isSpace . textOnlyHtml) h
      return t

```

`findTable` searches an HTML page for a table whose header satisfies the predicate.

6.5 Form Functions

```

extractForm :: HtmlElement → Maybe Form
extractForm (HtmlTag "form" fattrs kids)
  = Just $ Form action method $ treeExtractMap input $ htmlToTree kids
  where

```

```

action = fromMaybe "" ("action" `lookupAttr` fattrs)
method = fromMaybe "get" ("method" `lookupAttr` fattrs)
input (Tree (HtmlTag "input" attrs _) _)
  | null name = Nothing
  | kind == "checkbox" = Just $ FormItem name (valueIf checked) [value]
  | kind == "radio" = Just $ FormItem name (valueIf selected) [value]
  | kind `elem` ["submit","image"] = Just $ FormItem name "" [value]
  | kind `elem` ["text","password","hidden",""] = Just $ FormItem name value []
  | otherwise = Nothing
where

```

```

kind = maybe "" (map toLower) ("type" `lookupAttr` attrs)
name = fromMaybe "" ("name" `lookupAttr` attrs)
defVal | kind == "checkbox" = "YES"
      | otherwise = ""
value = fromMaybe defVal ("value" `lookupAttr` attrs)
checked = isJust ("checked" `lookupAttr` attrs)
selected = isJust ("selected" `lookupAttr` attrs)
valueIf p = if p then value else ""

```

```

input me@(Tree (HtmlTag "select" attrs _) _)
  | null name = Nothing
  | not (null options) = Just $ FormItem name value choices
where
  name = fromMaybe "" ("name" `lookupAttr` attrs)
  options = treeExtractMap option me
  value = snd $ fromMaybe (head options) $ find fst options
  choices = map snd options

```

```

input _ = Nothing
option (Tree (HtmlTag "option" attrs _) _) = Just (selected,value)
  where
    selected = isJust ("selected" `lookupAttr` attrs)
    value = fromMaybe "" ("value" `lookupAttr` attrs)
option _ = Nothing
extractForm _ = Nothing

```

```

lookupFormItem :: String → Form → Maybe FormItem
lookupFormItem k = find ((==k).formItemName) . formItems

```

```

updateFormValue :: String → String → Form → Form
updateFormValue k v frm = frm{formItems=map f (formItems frm)}
  where
    f fi | formItemName fi == k = fi{formItemValue=v}
    | otherwise = fi

```

```
urlEncodeForm :: Form -> String
urlEncodeForm (Form _ _ xs) = concat $ intersperse "&" items
  where
    grouped = groupByField formItemName $ sortByField formItemName xs
    items = catMaybes (map mkItem grouped)
    mkItem xs
      | null val = Nothing
      | otherwise = Just $ formItemName (head xs) ++ "=" ++ val
    where val = concat
              $ intersperse ","
              $ map (urlEncode.formItemValue) xs
```

Chapter 7

Network Libraries

7.1 CURL Bindings

7.1.1 Introduction

This is an easy to used Haskell interface to CURL. It exposes a very low level interface to CURL that isn't properly type checked. It *is* possible crash your application, cause buffer overflows, etc using this interface. It is recommended that most users stay away from the `curlEasySetOpt` function unless they know what they're going. `curlEasyInit` and the `curlSimple` family should be enough for most people.

For an even more high level interface see `Brianweb.Net.WebBrowser`

7.1.2 Foreign Imports

```
foreign import ccall "curl/curl.h curl_easy_init"
  curl_easy_init :: IO (Ptr ())
foreign import ccall "curl/curl.h curl_easy_perform"
  curl_easy_perform :: Ptr () → IO Int
foreign import ccall "curl/curl.h &curl_easy_cleanup"
  p_curl_easy_cleanup :: FunPtr (Ptr () → IO ())
```

```
foreign import ccall "Brianweb/Net/CurlHelper.h curlhelper_easy_setopt_long"
  curl_easy_setopt_long :: Ptr () → Int → CLong → IO Int

foreign import ccall "Brianweb/Net/CurlHelper.h curlhelper_easy_setopt_objectpoint"
  curl_easy_setopt_objectpoint :: Ptr () → Int → Ptr () → IO Int

foreign import ccall "Brianweb/Net/CurlHelper.h curlhelper_easy_setopt_functionpoint"
  curl_easy_setopt_functionpoint :: Ptr () → Int → FunPtr () → IO Int
```

```
foreign import ccall "wrapper"
  mkCurlWriteCB :: CurlWriteCB → IO (FunPtr CurlWriteCB)
```

7.1.3 Curl Types

```
type CurlHandle = ForeignPtr ()  
type CurlWriteCB = Ptr CChar → CSize → CSize → Ptr () → IO CSize
```

```
data CurlCode  
= CurlCodeOk | CurlCodeUnsupportedProtocol | CurlCodeFailedInit | CurlCodeUrlMalformat  
| CurlCodeUrlMalformatUser | CurlCodeCouldntResolveProxy | CurlCodeCouldntResolveHost  
| CurlCodeCouldntConnect | CurlCodeFtpWeirdServerReply | CurlCodeFtpAccessDenied  
| CurlCodeFtpUserPasswordIncorrect | CurlCodeFtpWeirdPassReply | CurlCodeFtpWeirdUserReply  
| CurlCodeFtpWeirdPasvReply | CurlCodeFtpWeird | CurlCodeFtpCantGetHost  
| CurlCodeFtpCantReconnect | CurlCodeFtpCouldntSetBinary | CurlCodePartialFile  
| CurlCodeFtpCouldntRetrFile | CurlCodeFtpWriteError | CurlCodeFtpQuoteError  
| CurlCodeHttpNotFound | CurlCodeWriteError | CurlCodeMalformatUser  
| CurlCodeFtpCouldntStorFile | CurlCodeReadError | CurlCodeOutOfMemory  
| CurlCodeOperationTimeouted | CurlCodeFtpCouldntSetAscii | CurlCodeFtpPortFailed  
| CurlCodeFtpCouldntUseRest | CurlCodeFtpCouldntGetSize | CurlCodeHttpRequestError  
| CurlCodeHttpPostError | CurlCodeSslConnectError | CurlCodeBadDownloadResume  
| CurlCodeFileCouldntReadFile | CurlCodeLdapCannotBind | CurlCodeLdapSearchFailed  
| CurlCodeLibraryNotFound | CurlCodeFunctionNotFound | CurlCodeAbortedByCallback  
| CurlCodeBadFunctionArgument | CurlCodeBadCallingOrder | CurlCodeHttpPortFailed  
| CurlCodeBadPasswordEntered | CurlCodeTooManyRedirects | CurlCodeUnknownTelnetOption  
| CurlCodeTelnetOptionSyntax | CurlCodeObsolete | CurlCodeSslPeerCertificate  
| CurlCodeGotNothing | CurlCodeSslEngineNotfound | CurlCodeSslEngineSetfailed  
| CurlCodeSendError | CurlCodeRecvError | CurlCodeShareInUse | CurlCodeSslCertproblem  
| CurlCodeSslCipher | CurlCodeSslCacert | CurlCodeBadContentEncoding  
deriving (Show, Eq, Enum)
```

```
data CurlOption  
= CurlOptUnused0 | CurlOptFile | CurlOptUrl | CurlOptPort | CurlOptProxy | CurlOptUserpwd  
| CurlOptProxyuserpwd | CurlOptRange | CurlOptUnused8 | CurlOptInfile  
| CurlOptErrorbuffer | CurlOptWritefunction | CurlOptReadfunction | CurlOptTimeout  
| CurlOptInfilesize | CurlOptPostfields | CurlOptReferer | CurlOptFtpport  
| CurlOptUseragent | CurlOptLowSpeedLimit | CurlOptLowSpeedTime | CurlOptResumeFrom  
| CurlOptCookie | CurlOptHttpheader | CurlOptHttppost | CurlOptSslcert  
| CurlOptSslkeypasswd | CurlOptCrlf | CurlOptQuote | CurlOptWriteheader | CurlOptUnused30  
| CurlOptCookiefile | CurlOptSslversion | CurlOptTimecondition | CurlOptTimevalue  
| CurlOptHttprequest | CurlOptCustomrequest | CurlOptStderr | CurlOptUnused38  
| CurlOptPostquote | CurlOptWriteinfo | CurlOptVerbose | CurlOptHeader | CurlOptNoprogress  
| CurlOptNobody | CurlOptFailonerror | CurlOptUpload | CurlOptPost | CurlOptFtpplistonly  
| CurlOptUnused49 | CurlOptFtpappend | CurlOptNetrc | CurlOptFollowlocation  
| CurlOptTransfertext | CurlOptPut | CurlOptMute | CurlOptProgressfunction  
| CurlOptProgressdata | CurlOptAutoreferer | CurlOptProxyport | CurlOptPostfieldsize  
| CurlOptHttpproxytunnel | CurlOptInterface | CurlOptKrb4level | CurlOptSslVerifypeer  
| CurlOptCainfo | CurlOptPasswdfunction | CurlOptPasswddata | CurlOptMaxredirs  
| CurlOptFiletime | CurlOptTelnetoptions | CurlOptMaxconnects | CurlOptClosepolicy  
| CurlOptClosefunction | CurlOptFreshConnect | CurlOptForbidReuse | CurlOptRandomFile  
| CurlOptEgdssocket | CurlOptConnecttimeout | CurlOptHeaderfunction | CurlOptHttpget  
| CurlOptSslVerifyhost | CurlOptCookiejar | CurlOptSslCipherList | CurlOptHttpVersion  
| CurlOptFtpUseEpsv | CurlOptSslcerttype | CurlOptSslkey | CurlOptSslkeytype  
| CurlOptSslengine | CurlOptSslengineDefault | CurlOptDnsUseGlobalCache  
| CurlOptDnsCacheTimeout | CurlOptPrequote | CurlOptDebugfunction | CurlOptDebugdata  
| CurlOptCookiesession | CurlOptCapath | CurlOptBufferSize | CurlOptNosignal | CurlOptShare  
| CurlOptProxytype | CurlOptEncoding  
deriving (Show, Eq, Enum)
```

7.1.4 Helper Function

```
curlCheck :: IO CurlCode → IO ()
curlCheck action = do
  code ← action
  when (code ≠ CurlCodeOk) $
    fail $ "CURL Error Code: " ++ show code ++ " (" ++ show (fromEnum code) ++ ")"
  return ()
```

7.1.5 Curl Function Wrappers

```
curlEasyInit :: IO CurlHandle
curlEasyInit = do
  ptr ← curl_easy_init
  h ← if ptr == nullPtr
    then fail "curl_easy_init failed"
    else newForeignPtr p_curl_easy_cleanup ptr
  return h
```

```
curlEasySetOpt' :: (Ptr () → Int → a → IO Int) → CurlHandle → CurlOption → a → IO CurlCode
curlEasySetOpt' realFunc fptr opt arg = withForeignPtr fptr $
  λptr → fmap toEnum (realFunc ptr (fromEnum opt) arg)
```

```
class CurlEasySetOptValue a where
  curlEasySetOpt :: CurlHandle → CurlOption → a → IO CurlCode

instance CurlEasySetOptValue Bool where
  curlEasySetOpt = (.fromEnum) .! curlEasySetOpt

instance CurlEasySetOptValue Int where
  curlEasySetOpt = (.fromIntegral) .! curlEasySetOpt' curl_easy_setopt_long

instance CurlEasySetOptValue (Ptr a) where
  curlEasySetOpt = (.castPtr) .! curlEasySetOpt' curl_easy_setopt_objectpoint

instance CurlEasySetOptValue (FunPtr a) where
  curlEasySetOpt = (.castFunPtr) .! curlEasySetOpt' curl_easy_setopt_functionpoint
```

```
curlEasyPerform :: CurlHandle → IO CurlCode
curlEasyPerform fptr = withForeignPtr fptr (fmap toEnum . curl_easy_perform)
```

7.1.6 Curl Callbacks

```
curlWriteCB :: CurlWriteCB
curlWriteCB ptr size n userData = do
  let realsize = size*n
      buf ← deRefStablePtr (castPtrToStablePtr userData)
      bufAppendRaw buf ptr (fromIntegral realsize)
  return realsize
```

`curlWriteCB` a callback for `CurlOptWritefunction` and `CurlOptHeaderfunction`. It appends the data from `libcurl` to a `Buffer` supplied as a `StablePtr` in `userData`.

7.1.7 Simple Interface

```
curlSimpleGo :: CurlHandle → String → IO ([CChar],[CChar])
curlSimpleGo h url = do
  withCString url $ \urlPtr → do
    curlCheck $ curlEasySetOpt h CurlOptUrl urlPtr

    bracket (mkCurlWriteCB curlWriteCB) freeHaskellFunPtr $ \fptr → do

      buf ← newBuffer
      bracket (newStablePtr buf) freeStablePtr $ \stableBuf → do
        curlCheck $ curlEasySetOpt h CurlOptWritefunction fptr
        curlCheck $ curlEasySetOpt h CurlOptFile (castStablePtrToPtr stableBuf)

      headerBuf ← newBuffer
      bracket (newStablePtr headerBuf) freeStablePtr $ \stableHeaderBuf → do
        curlCheck $ curlEasySetOpt h CurlOptHeaderfunction fptr
        curlCheck $ curlEasySetOpt h CurlOptWriteheader (castStablePtrToPtr stableHeaderBuf)

      curlCheck (curlEasyPerform h)

      docData ← bufElems buf
      headerData ← bufElems headerBuf
      return (docData,headerData)
```

`curlSimpleGo` sets `CurlOptUrl`, `CurlOptWritefunction`, `CurlOptHeaderfunction`, `CurlOptFile`, and `CurlOptWriteheader` before calling `curlEasyPerform`. It then extracts the body and headers from the response and returns them as `[CChar]`s.

```
curlSimpleGet :: CurlHandle → String → IO ([CChar],[CChar])
curlSimpleGet h url = do
  curlCheck $ curlEasySetOpt h CurlOptHttpget True
  curlSimpleGo h url

curlSimpleGetText :: CurlHandle → String → IO String
curlSimpleGetText h u = fmap (map castCCharToChar . fst) $ curlSimpleGet h u
```

```
curlSimplePost :: CurlHandle → String → String → IO ([CChar],[CChar])
curlSimplePost h u post = do
  curlCheck $ curlEasySetOpt h CurlOptPost True
  withCString post $ \postPtr → do
    curlCheck $ curlEasySetOpt h CurlOptPostfields postPtr
    curlSimpleGo h u

curlSimplePostText :: CurlHandle → String → String → IO String
curlSimplePostText h u p = fmap (map castCCharToChar . fst) $ curlSimplePost h u p
```

7.2 HTTP Protocol Utilities

7.2.1 Introduction

This module contains a few functions for dealing with the HTTP protocol, mainly parsing headers. It is *not* an HTTP client. See `Brianweb.Net.Curl` and `Brianweb.Net.WebBrowser` for that.

7.2.2 Header Parsing

```
parseHTTPStatusAndHeaders :: String → Either ParseError (Int, [(String, String)])
parseHTTPStatusAndHeaders = runParser (followedBy eof parseHTTPStatusAndHeaders') () ""

parseHTTPStatusAndHeaders' :: Parser (Int, [(String, String)])
parseHTTPStatusAndHeaders' = do
  (_, code, _) ← parseHTTPStatus
  headers ← parseHTTPHeaders'
  return (code, headers)
```

```
parseHTTPStatus :: Parser ((Int, Int), Int, String)
parseHTTPStatus = do
  string "HTTP/"
  major ← many1 digit >>= readIntegral
  char '.'
  minor ← many1 digit >>= readIntegral
  many1 (oneOf " \t")
  status ← count 3 digit >>= readIntegral
  spaces
  msg ← manyTill anyChar (many1 $ oneOf "\r\n")
  return ((major, minor), status, msg)
```

```
parseHTTPHeaders :: String → Either ParseError [(String, String)]
parseHTTPHeaders = runParser (followedBy eof parseHTTPHeaders') () ""

parseHTTPHeaders' :: Parser [(String, String)]
parseHTTPHeaders' = many parseHTTPHeader
```

```
parseHTTPHeader :: Parser (String, String)
parseHTTPHeader = do
  name ← fmap (rightTrim isSpace)
    $ manyTill (noneOf "\r\n") (char ':')
  many (oneOf " \t")
  value ← fmap (concat.intersperse " ")
    $ (manyTill anyChar (many1 $ oneOf "\r\n")) `sepBy` (many1 $ oneOf " \t")
  return (name, value)
```

7.3 URI Functions

7.3.1 Introduction

This module contains function for parsing, showing, and qualifying URIs.

7.3.2 Data Types

```
data URI
  = AbsoluteURI {
    uriScheme :: String,
    uriAuthority :: Maybe URIAuthority,
    uriPath :: String,
    uriQuery :: Maybe String,
    uriReference :: Maybe String
  }
  | RelativeURI {
    uriPath :: String,
    uriQuery :: Maybe String,
    uriReference :: Maybe String
  }
deriving (Eq)
```

```
data URIAuthority = URIAuthority {
  uriAuthUser :: Maybe String,
  uriAuthHost :: String,
  uriAuthPort :: Maybe Integer
} deriving (Eq)
```

7.3.3 Instance Declarations

```
instance Show URI where
  showsPrec _ = showsURI
```

7.3.4 Public API

```
parseURI :: String → Either ParseError URI
parseURI = runParser parseURI' () ""
```

```
isAbsoluteURI :: URI → Bool
isAbsoluteURI (AbsoluteURI {}) = True
isAbsoluteURI _ = False
```

```

makeAbsoluteURI :: (Maybe URI) → URI → Maybe URI
makeAbsoluteURI _ uri@(AbsoluteURI {}) = Just uri
makeAbsoluteURI (Just (prev@(AbsoluteURI {}))) (RelativeURI p q r)
  = Just prev{uriPath=p',uriQuery=q,uriReference=r}
  where
    p' = case p of
      ('/':_) → p
      _ → rightTrim (≠'/') p' ++ p'
makeAbsoluteURI _ _ = Nothing

```

makeAbsoluteURI tries to convert a URI to an absolute URI given knowledge of the “current” URI. If the new URI is already absolute it is returned as is. If it is not absolute but we have a previous uri that it qualifies the new URI using the previous URI.

```

urlEncode :: String → String
urlEncode = ($) . escapeString ""

```

7.3.5 Show Functions

```

showsAuthority :: URIAuthority → ShowS
showsAuthority (URIAuthority a h p)
  = maybe id (λx → (x++) . ('@':)) a
  . (h++)
  . maybe id (λx → (':':) . (shows x)) p

```

```

showsURI :: URI → ShowS
showsURI (AbsoluteURI s a p q r)
  = (s++)
  . (':':)
  . maybe id (λx → ("//"++) . showsAuthority x) a
  . (escapeString ":@&=+$/ " p)
  . maybe id (λx → ('?':) . escapeString ";/?:@&=+$/ " x) q
  . maybe id (λx → ('#':) . escapeString ";/?:@&=+$/ " x) r
showsURI (RelativeURI p q r)
  = (escapeString ":@&=+$/ " p)
  . maybe id (λx → ('?':) . escapeString ";/?:@&=+$/ " x) q
  . maybe id (λx → ('#':) . escapeString ";/?:@&=+$/ " x) r

```

```

escapeString :: String → String → ShowS
escapeString allowed s z = foldr f z s
  where
    f c | isAlphaNum c || c `elem` "-_!.~*()" || c `elem` allowed = (c:)
        | otherwise = ('%':) . (intToDigit (x `quot` 16)) . (intToDigit (x `rem` 16))
        where x = ord c `rem` 256

```

7.3.6 Parser

```

parseURI' :: Parser URI
parseURI' = followedBy eof $ try parseAbsoluteURI <|> parseRelativeURI

```

```

parseAbsoluteURI :: Parser URI
parseAbsoluteURI = do
  scheme ← parseScheme
  char ':'
  authority ← parseAuthority
  path ← char '/' >> parsePathSegment `sepBy` (char '/')
    >>= return.('/:').concat.intersperse "/"
  query ← parseQuery
  reference ← parseReference
  return $ AbsoluteURI scheme authority path query reference

```

```

parseRelativeURI :: Parser URI
parseRelativeURI = do
  path ← parsePathSegment `sepBy` (char '/') >>= return.concat.intersperse "/"
  query ← parseQuery
  reference ← parseReference
  return $ RelativeURI path query reference

```

```

parseQuery :: Parser (Maybe String)
parseQuery = maybeParse (char '?' >> many (escapedChar <|> oneOf ";/?:@&=+,$"))

```

```

parseReference :: Parser (Maybe String)
parseReference = maybeParse (char '#' >> many (escapedChar <|> oneOf ";/?:@&=+,$"))

```

```

parsePathSegment :: Parser String
parsePathSegment = followedBy (optional $ char ':' >> many pchar) (many pchar)
  where
    pchar = escapedChar <|> oneOf ":@&=+,$"

```

```

parseScheme :: Parser String
parseScheme = do
  c ← letter
  cs ← many (alphaNum <|> oneOf "+-." )
  return (c:cs)

```

```

escapedChar :: Parser Char
escapedChar = alphaNum <|> oneOf "-_!.~*()" <|> eseq
  where
    eseq = do
      char '%'
      d1 ← hexDigit
      d2 ← hexDigit
      return $ chr $ (digitToInt d1 * 16 + digitToInt d2)

```

```

parseAuthority :: Parser (Maybe URIAuthority)
parseAuthority = maybeParse $ do
  count 2 (char '/')
  userInfo ← maybeParse $ try $ followedBy (char '@') (many (escapedChar <|> oneOf ";&=+,$"))
  host ← many1 (alphaNum <|> oneOf "-.")
  port ← maybeParse $ char ':' >> many1 digit >>= readIntegral
  return $ URIAuthority userInfo host port

```

7.4 Web Browser

7.4.1 Introduction

This module is a very high level interface to `Brianweb.Net.Curl`. It handles maintaining state so relative URIs can be used, parsing HTML, posting form values, handling HTTP redirects, etc. When used in conjunction with `Brianweb.Text.Html` it is ideal for screen scraping.

7.4.2 Data Types

```
newtype WBState = WBState {  
  currentURI :: Maybe URI  
}  
  
data WebBrowser = WebBrowser CurlHandle (IORef WBState)
```

```
defaultState :: WBState  
defaultState = WBState Nothing
```

7.4.3 Public API

```
newWebBrowser :: IO WebBrowser  
newWebBrowser = do  
  h ← curlEasyInit  
  —curlCheck $ curlEasySetOptLong h CurlOptVerbose 1  
  curlCheck $ curlEasySetOpt h CurlOptCookiefile emptyCString  
  r ← newIORef defaultState  
  return (WebBrowser h r)
```

```
loadPage :: WebBrowser → String → (Maybe String) → IO String  
loadPage self@(WebBrowser curl stateRef) req post = do  
  last ← fmap currentURI (readIORef stateRef)  
  new' ← fromEitherM (parseURI req)  
  new ← maybe (fail "need an absolute uri here") return  
    $ makeAbsoluteURI last new'  
  (body,rawHeaders) ← case post of  
    Just x → curlSimplePost curl (show new) x  
    Nothing → curlSimpleGet curl (show new)  
  (_,headers) ← fromEitherM $ parseHTTPStatusAndHeaders $ map castCCharToChar rawHeaders  
  modifyIORef stateRef (\s → s{currentURI=Just new})  
  case "Location" `lookup` headers of  
    Just newLocation → loadPage self newLocation Nothing  
    Nothing → return $ (map castCCharToChar) body
```

```
loadHtmlPage :: WebBrowser → String → (Maybe String) → IO Html  
loadHtmlPage wb uri post = loadPage wb uri post >>= fromEitherM . parseHtmlDoc uri
```

```

submitForm :: WebBrowser → Form → IO Html
submitForm self form
  | method == "post" = do
    loadHtmlPage self (formAction form) (Just encoded)
  | otherwise = do
    url ← fromEitherM $ parseURI $ formAction form
    loadHtmlPage self (show (url{uriQuery=Just encoded})) Nothing
  where
    method = map toLower (formMethod form)
    encoded = urlEncodeForm form

```

7.5 Misc IO Functions

```

cat :: [String] → IO (String)
cat [] = hGetContents stdin
cat l = do
  chunks ← cat' l
  return $ concat chunks
  where
    cat' [] = return []
    cat' (x:xs) = do
      s ← unsafeInterleaveIO $ readFile x
      rest ← cat' xs
      return $ s:rest

```

cat acts somewhat like the standard UNIX cat command. It takes a list of files and returns the contents of all those files concatenated together. If no files are specified `stdin` is read.

```

keyPrompt :: Handle → Handle → String → [Char] → IO (Char)
keyPrompt inh outh prompt choices = do
#ifdef __GLASGOW_HASKELL__
  isterm ← hIsTerminalDevice inh
  c ← if isterm
    then do
      hPutStr outh prompt
      hFlush outh
      ob ← hGetBuffering inh
      oe ← hGetEcho inh
      hSetBuffering inh NoBuffering
      hSetEcho inh False
      c2 ← hGetChar inh
      hSetEcho inh oe
      hSetBuffering inh ob
      hPutStrLn outh ""
      return c2
    else do
      #else
      c ← do
      #endif
      hPutStrLn outh prompt
      hGetLine inh >>= return . head . (++"??")
      if elem (toLower c) choices
        then return $ toLower c
        else do
          hPutStrLn outh $ [c] ++ " is not a valid choice"
          keyPrompt inh outh prompt choices

```

keyPrompt prompts the user to press a key from a specified list of keys.

```
hGetContentsStrict :: Handle → IO [Char]
hGetContentsStrict h = loop
  where
    loop = do
      eof ← hIsEOF h
      if eof
        then return []
        else do
          c ← hGetChar h
          cs ← loop
          return (c:cs)

readFileStrict :: String → IO [Char]
readFileStrict fn = bracket (openFile fn ReadMode) hClose hGetContentsStrict
```

7.6 Foreign Function Interface Utilities

```
{-# NOINLINE emptyCString #-}
emptyCString :: CString
emptyCString = unsafePerformIO $ newCString ""
```

Chapter 8

Type Level Magic

8.1 Type Level Naturals

8.1.1 Introduction

This module implements type-level naturals and many operations on them as described in [KLS04]. It can be used to create lists whose length is known by the type system, strongly typed operations on fixed-sized words (see `Brianweb.Word`), etc. One particularly important feature of them is they are implemented in such a way that their value is not only known by the type system but can be turned into a first-class value at compile time. This allows things like type-safe words to be implemented very efficiently.

8.1.2 Data and Class Declarations

```
data TZero
data TSucc n
```

The type-level equivalent of `data Nat = Zero | Succ Nat`

```
class TNat n where
  tNatFold :: (a -> a) -> a -> n -> a
instance TNat TZero where
  tNatFold _ z _ = z
instance TNat a => TNat (TSucc a) where
  tNatFold f z = f . tNatFold f z . a
  where a :: TSucc a -> a
        a = undefined
```

The `TNat` class with a fold operation for converting to a first class integer, mask, etc. This fold can usually be optimized away by the compiler.

```

data TTrue
data TFalse

class TBool a
instance TBool TTrue
instance TBool TFalse

```

The type-level equivalent of `data Bool = True | False`

8.1.3 Operations

```

class (TNat a, TNat b, TNat c) => TNatAdd a b c | a b -> c
instance TNat b => TNatAdd TZero b b
instance TNatAdd a b c => TNatAdd (TSucc a) b (TSucc c)

```

Type-level addition. `a` and `b` are the operands, `c` is the result.

```

class (TNat a, TNat b, TNat c) => TNatSub a b c | a b -> c
instance TNat a => TNatSub a TZero a
instance TNatSub a b c => TNatSub (TSucc a) (TSucc b) c

```

Type-level subtraction. `a` and `b` are the operands, `c` is the result.

```

class TNatMul a b c | a b -> c
instance TNat a => TNatMul a TZero TZero
instance (TNatMul a m c, TNatAdd a c p) => TNatMul a (TSucc m) p

```

Type-level multiplication. `a` and `b` are the operands, `c` is the result.

```

class (TNat a, TNat b, TNat c) => TNatDiv a b c | a b -> c
instance TNat x => TNatDiv TZero x TZero
instance (TNatSub x d x', TNatDiv x' d q, TNatAdd q (TSucc TZero) q') => TNatDiv x d q'

```

Type-level division. `a` and `b` are the operands,

```

class (TNat a, TNat b, TBool c) => TNatLE a b c | a b -> c
instance TNatLE TZero TZero TTrue
instance TNat a => TNatLE (TSucc a) TZero TFalse
instance TNat a => TNatLE TZero (TSucc a) TTrue
instance (TNat a, TNat b, TNatLE a b c) => TNatLE (TSucc a) (TSucc b) c

```

Type-level comparison.

```

class (TNat a, TNat b, TBool c) => TNatLT a b c | a b -> c
instance TNatLT TZero TZero TFalse
instance TNat a => TNatLT (TSucc a) TZero TFalse
instance TNat a => TNatLT TZero (TSucc a) TTrue
instance (TNat a, TNat b, TNatLT a b c) => TNatLT (TSucc a) (TSucc b) c

```

Type-level comparison.

8.1.4 Real Functions

These functions are mainly for testing. They are all `undefined` so they are only useful for their type.

```
tNatAdd :: TNatAdd a b c => a -> b -> c
tNatAdd = undefined
```

```
tNatSub :: TNatSub a b c => a -> b -> c
tNatSub = undefined
```

```
tNatMul :: TNatMul a b c => a -> b -> c
tNatMul = undefined
```

```
tNatDiv :: TNatDiv a b c => a -> b -> c
tNatDiv = undefined
```

8.2 Strongly Typed Heterogeneous Lists

8.2.1 Introduction

This module provides functions for working on strongly typed heterogeneous lists. The ideas here (and a good chunk of the code) are based heavily on [KLS04].

8.2.2 Data and Class Declarations

```
data HNil = HNil deriving Show
data HCons h t = HCons h !t deriving Show

infixr 5 `HCons`
```

```
class HList a
instance HList HNil
instance HList t => HList (HCons h t)
```

8.2.3 Operations

```
class (TNat n, HList l)  $\Rightarrow$  HLength l n | l  $\rightarrow$  n
instance HLength HNil TZero
instance HLength t n  $\Rightarrow$  HLength (HCons h t) (TSucc n)

hLength :: HLength l n  $\Rightarrow$  l  $\rightarrow$  Int
hLength = tNatFold (+1) 0 . hLength'
  where
    hLength' :: HLength l n  $\Rightarrow$  l  $\rightarrow$  n
    hLength' = undefined
```

```
class HList l  $\Rightarrow$  HomogeneousHList l a
instance HomogeneousHList HNil a
instance HomogeneousHList t a  $\Rightarrow$  HomogeneousHList (HCons a t) a
```

```
class (HList l, HList l')  $\Rightarrow$  HMap a b l l' | l b  $\rightarrow$  l', l' a  $\rightarrow$  l where
  hMap :: (a  $\rightarrow$  b)  $\rightarrow$  l  $\rightarrow$  l'

instance HMap a b HNil HNil where
  hMap _ _ = HNil

instance HMap a b t t'  $\Rightarrow$  HMap a b (HCons a t) (HCons b t') where
  hMap f (HCons h t) = HCons (f h) (hMap f t)
```

```
class (HList l, HList l', HList l'')  $\Rightarrow$  HZip a b c l l' l''
  | l l' c  $\rightarrow$  l'', l'' a  $\rightarrow$  l, l'' b  $\rightarrow$  l' where
  hZip :: (a  $\rightarrow$  b  $\rightarrow$  c)  $\rightarrow$  l  $\rightarrow$  l'  $\rightarrow$  l''

instance HZip a b c HNil HNil HNil where
  hZip _ _ _ = HNil

instance HZip a b c t t' t''  $\Rightarrow$  HZip a b c (HCons a t) (HCons b t') (HCons c t'') where
  hZip f (HCons h t) (HCons h' t') = HCons (f h h') (hZip f t t')
```

Chapter 9

Type-Safe Words

9.1 Introduction

This module implements Type-Safe fixed length words. This allows words of any size (8-bit, 16-bit, 2-bit, 13-bit, etc) to be manipulated and combined in a type-safe and efficient way. One of the most important features of this module is that all the operations compile down to pretty much exactly what would have been coded manually. The type-safety is free.

9.2 Word Types

```
newtype Word n b = W b
```

A `Word` is just a “native word” with a bit length attached to it (as a type-level natural).

```
— FEATURE: Should check if it really needs to do mask (no need to and with 0xffffffff)  
w :: (TNat n, WordBase b) => b -> Word n b  
w x = r where r = W (x DB.&. wordMask r)
```

`w` is the “safe” constructor. It ensures the native value is within the range of the word type. This will compile down to an “and” operation when it is used.

9.2.1 Base Type

```
class (Integral a, DB.Bits a) => WordBase a where  
  
instance WordBase DW.Word  
instance WordBase DW.Word32  
instance WordBase DW.Word64
```

All words have a “base type”. That is the actual native type that represents a word. There are only a few word sizes available in Haskell.

```
— FEATURE: Can we be smarter here?
wordBaseCast :: (WordBase a, WordBase b) => a -> b
wordBaseCast = fromIntegral
```

```
asSize :: BaseType n b => Word n b -> n -> Word n b
asSize x _ = x
```

9.3 Operations

9.3.1 Word Info

```
wordSize :: TNat n => Word n b -> Int
wordSize = tNatFold (+1) 0 . n
  where n :: (Word n b) -> n
        n = undefined

wordMask :: (TNat n, WordBase b) => Word n b -> b
wordMask = tNatFold (\x -> x*2 + 1) 0 . n
  where n :: (Word n b) -> n
        n = undefined
```

These functions find the word size and mask word. Both these functions will be completely eliminated at compile-time.

9.3.2 Casts

```
unsafeWordCast :: (WordBase b1, WordBase b2) => (Word n1 b1) -> (Word n2 b2)
unsafeWordCast (W x) = W (wordBaseCast x)
```

`unsafeWordCast` casts a word from one size to another. It **does not** truncate the word when converting from a larger to a smaller word. It is up to the caller to prove the value is within the valid range of the destination type. In most cases this will compile down to “id” at compile time.

```
wordCast :: (TNat n1, TNat n2, WordBase b1, WordBase b2) => (Word n1 b1) -> (Word n2 b2)
wordCast a@(W x) = b where b = (if wordSize b < wordSize a then w else W) (wordBaseCast x)
```

`wordCast` is a safe version of the above function.

9.3.3 Lifting Functions

```
liftW :: (b → a) → Word n b → a
liftW f (W x) = f x

liftW2 :: (b → b → a) → Word n b → Word n b → a
liftW2 f (W x) (W y) = f x y

liftW3 :: (b → b → b → a) → Word n b → Word n b → Word n b → a
liftW3 f (W x) (W y) (W z) = f x y z
```

9.3.4 Bitwise Operations

```
(.&.), (.|.), (.^.) :: WordBase b ⇒ Word n b → Word n b → Word n b
(&.) = (W.) . liftW2 (DB.&.)
(.|.) = (W.) . liftW2 (DB.|.)
(.^.) = (W.) . liftW2 DB.xor

(>>.), unsafeWordSHL :: WordBase b ⇒ (Word n b) → Int → (Word n b)
(W x) .>>. y = W (x `DB.shiftR` y)
(W x) `unsafeWordSHL` y = W (x `DB.shiftL` y)

(<<.) :: (TNat n, WordBase b) ⇒ Word n b → Int → Word n b
(W x) .<<. y = w (x `DB.shiftL` y)

testBit :: (TNat n, WordBase b) ⇒ Word n b → Int → Bool
testBit t@(W x) n = DB.testBit x (wordSize t - n)

bit :: (TNat n, WordBase b) ⇒ Int → Word n b
bit n = r where r = W (DB.bit (wordSize r - n))

rotr, rotl :: (TNat n, WordBase b) ⇒ (Word n b) → Int → (Word n b)
x `rotr` y = (x .>>. y) .|. (x .<<. (wordSize x - y))
x `rotl` y = (x .<<. y) .|. (x .>>. (wordSize x - y))

— FEATURE: setBit, clearBit, complement
```

9.4 Standard Typeclass Instances

```
typeString :: TNat n ⇒ (Word n b) → String
typeString x = "Word" ++ show (wordSize x :: Int)
```

9.4.1 Bounded

```
instance (TNat n, WordBase b) ⇒ Bounded (Word n b) where
  minBound = W 0
  maxBound = r where r = W (wordMask r)
```

The minBound of an Word is always 0 (they are unsigned). The maxBound is the mask of the word.

9.4.2 Enum

```

succError :: TNat n => (Word n b) -> a
succError x = error $ "Enum.succ{" ++ typeString x ++
                    "}: tried to take 'succ' of maxBound"

predError :: TNat n => (Word n b) -> a
predError x = error $ "Enum.succ{" ++ typeString x ++
                    "}: tried to take 'pred' of minBound"

```

```

toEnumError :: (WordBase b, TNat n) => Int -> (Word n b)
toEnumError x = r where
  r = error $ "Enum.toEnum{" ++ typeString r ++ "}: tag (" ++
        show x ++ ") is outside bounds (0," ++
        show (wordMask r) ++ ")"

fromEnumError :: (WordBase b, TNat n) => (Word n b) -> a
fromEnumError x = error $ "Enum.fromEnum{" ++ typeString x ++ "}: value (" ++
        show x ++ ") is outside Int's bounds " ++
        show (minBound::Int, maxBound::Int)

```

```

instance (TNat n, WordBase b) => Enum (Word n b) where
  succ x'@(W x)
    | x' ≠ maxBound = W (x + 1)
    | otherwise     = succError x'
  pred x'@(W x)
    | x' ≠ minBound = W (x - 1)
    | otherwise     = predError x'
  toEnum i = r where
    -- FIXME: Do an .&. instead
    r | i ≥ 0 && i ≤ fromIntegral (wordMask r) = W (fromIntegral i)
      | otherwise = toEnumError i
  fromEnum x'@(W x)
    | wordSize x' > DB.bitSize (undefined::Int) && x > fromIntegral (maxBound::Int)
    = fromEnumError x'
    | otherwise = fromIntegral x
  enumFrom x = enumFromTo x maxBound
  enumFromThen x y
    | x ≤ y = enumFromThenTo x y maxBound
    | otherwise = enumFromThenTo x y minBound
  enumFromTo (W x) (W y) = map W [x..y]
  enumFromThenTo (W x) (W y) (W z) = map W [x,y..z]

```

9.4.3 Eq

```

instance Eq b => Eq (Word n b) where
  (==) = liftW2 (==)
  (/=) = liftW2 (/=)

```

9.4.4 Integral

```
instance (TNat n, WordBase b) => Integral (Word n b) where
  quot = (w.) . liftW2 quot
  div = (w.) . liftW2 div
  rem = (w.) . liftW2 rem
  mod = (w.) . liftW2 mod
  (W x) 'quotRem' (W y) = (W q, W r) where
    (q,r) = (x 'quotRem' y)
  (W x) 'divMod' (W y) = (W q, W r) where
    (q,r) = (x 'divMod' y)
  toInteger (W x) = toInteger x
```

9.4.5 Ix

```
indexError :: (TNat n, WordBase b) => (Word n b) -> (Word n b, Word n b) -> a
indexError i b = error $ "Ix.index{" ++ typeString i ++ "}: Index " ++
  show i ++ " out of bounds " ++ show b
```

```
instance (TNat n, Ix b) => Ix (Word n b) where
  range = map W . uncurry (liftW2 (curry range))
  index = uncurry (liftW3 (curry index))
  inRange = uncurry (liftW3 (curry inRange))
#ifdef __GLASGOW_HASKELL__FIXME
  unsafeIndex b@(a,_) = fromIntegral (i - a)
  unsafeRangeSize (l,h) = h - l + 1
#endif
```

9.4.6 Num

```
instance (TNat n, WordBase b) => Num (Word n b) where
  (+) = (w.) . liftW2 (+)
  (-) = (w.) . liftW2 (-)
  (*) = (w.) . liftW2 (*)
  negate = w . liftW negate
  abs = id
  signum 0 = 0
  signum _ = 1
  fromInteger = w . fromInteger
```

9.4.7 Ord

```
instance Ord b => Ord (Word n b) where
  (<) = liftW2 (<)
  (<=) = liftW2 (<=)
  (>) = liftW2 (>)
  (>=) = liftW2 (>=)
```

9.4.8 Real

```
instance (TNat n, WordBase b) => Real (Word n b) where
  toRational x = fromIntegral x % 1
```

9.4.9 Show

```
instance Integral b => Show (Word n b) where
  showsPrec _ (W x) = ('0':) . ('x':) . showHex x
```

9.5 Advanced Operations

9.5.1 Base Type Map

```
class BaseType n a | n -> a
```

In order for the type system to construct a word of an arbitrary size we need a way to map word sizes to base types. The `BaseType` class does this.

```
instance BaseType (TSucc N32) DW.Word64
instance BaseType (TSucc N30) DW.Word32
instance BaseType TZero      DW.Word
instance BaseType n a => BaseType (TSucc n) a
```

This code works but requires `-fallow-overlapping-instances`. For now, however, we're using a static list mapping each individual size to a base type (not shown). It would be nice if there was a way to do this without `-fallow-overlapping-instances`.

9.5.2 Join

```
class WordJoin a b | a -> b where
  wordJoin :: a -> b
```

`wordJoin` join's an `HList` of words in a type-safe manner. The size of the result is always the sum of the sizes of the inputs.

```
instance WordJoin HNil Word0 where
  wordJoin _ = W 0
```

The base case. Joining 0 words away results in a word of size 0 with value 0. This useless value will be optimized away at compile time.

```

instance (
  WordJoin t (Word n2 b2),
  TNatAdd n1 n2 n3,
  BaseType n3 b3,
  TNat n1, WordBase b1, TNat n2, WordBase b2, TNat n3, WordBase b3
) => WordJoin (HCons (Word n1 b1) t) (Word n3 b3) where
wordJoin (HCons h t)
  | wordSize t' == 0 = unsafeWordCast h
  | otherwise = (unsafeWordCast h `unsafeWordSHL` wordSize t') .|. unsafeWordCast t'
where
  t' = wordJoin t

```

The inductive case. Joining the head of a list of words with it's tail requires shifting the head left by the size of the tail and oring with the tail. When the tail is of size 0 these optimizations are eliminated.

9.5.3 Split

```

class WordSplit a b | b -> a where
  wordSplit :: a -> b

```

`wordSplit` splits a single word into an `HList` of many word of possibly different sizes.

```

instance WordSplit Word0 HNil where
  wordSplit _ = HNil

```

The base case. Splitting a word of size 0 always results in the empty list.

```

instance (
  WordSplit (Word n3 b3) t,
  TNatAdd n2 n3 n1,
  BaseType n1 b1,
  TNat n1, WordBase b1, TNat n2, WordBase b2, TNat n3, WordBase b3
) => WordSplit (Word n1 b1) (HCons (Word n2 b2) t) where
wordSplit x = HCons h (wordSplit t)
where
  h | wordSize t == 0 = unsafeWordCast x
    | otherwise = unsafeWordCast (x .>>. wordSize t)
  t = unsafeWordCast x .&. W (wordMask t)

```

The inductive case. Splitting a word into the head and tail of an `HList` requires shifting off the tail to get the head then masking off the head to get the tail. Again tails of size 0 are special cased.

9.5.4 Simple Join and Split Function

Dealing with `HList`'s for only 2 or 3 words is a pain. These functions do that for you.

```
wordJoin2 :: WordJoin (a `HCons` b `HCons` HNil) c => a -> b -> c
wordJoin2 a b = wordJoin (a `HCons` b `HCons` HNil)

wordJoin3 :: WordJoin (a `HCons` b `HCons` c `HCons` HNil) d => a -> b -> c -> d
wordJoin3 a b c = wordJoin (a `HCons` b `HCons` c `HCons` HNil)
```

```
wordSplit2 :: WordSplit a (b `HCons` c `HCons` HNil) => a -> (b,c)
wordSplit2 x = let (b `HCons` c `HCons` HNil) = wordSplit x in (b,c)

wordSplit3 :: WordSplit a (b `HCons` c `HCons` d `HCons` HNil) => a -> (b,c,d)
wordSplit3 x = let (b `HCons` c `HCons` d `HCons` HNil) = wordSplit x in (b,c,d)
```

9.6 Type Aliases

Word (TSucc (TSucc (TSucc (TSucc TZero)))) Data.Word.Word is no fun to type. (And that's just Word4, imagine Word64). So we provide aliases for most commonly used typed. You are not limited to these aliases though. You can still join a Word1 and a Word10 to get a what would be Word11 had it been listed here.

```
type N0 = TZero
type Word0 = Word N0 DW.Word
type N1 = TSucc N0; type Word1 = Word N1 DW.Word
type N2 = TSucc N1; type Word2 = Word N2 DW.Word
type N3 = TSucc N2; type Word3 = Word N3 DW.Word
type N4 = TSucc N3; type Word4 = Word N4 DW.Word
type N5 = TSucc N4; type Word5 = Word N5 DW.Word
type N6 = TSucc N5; type Word6 = Word N6 DW.Word
type N7 = TSucc N6; type Word7 = Word N7 DW.Word
type N8 = TSucc N7; type Word8 = Word N8 DW.Word
```

(Instances up to Word64 not shown.)

Chapter 10

Miscellaneous

10.1 Debugging Function

```
traceM :: Monad m => String -> m ()
traceM s = trace s $ return ()
```

`traceM` is useful for “printf style” debugging. With most monads (but not all) the debugging info will be displayed when that part of the monadic computation is evaluated.

10.2 Misc Helper Functions

```
— FIXME: Functor class (and maybe a new MonadFunctor class, fmapM?)
tupleMap :: (a -> b) -> (a,a) -> (b,b)
tupleMap f (a,b) = (f a, f b)

tupleMapM :: Monad c => (a -> c b) -> (a,a) -> c (b,b)
tupleMapM f (a,b) = do
  a' ← f a
  b' ← f b
  return (a',b')
```

`tupleMap` applies a function to both elements of a two-element tuple.

```

partialLookup :: Monad a => String -> [(String,b)] -> String -> a b
partialLookup key l noun =
  case filter ((key==) . take (length key) . fst) l of
    [x] -> return $ snd x
    [] -> fail $
      " Unknown " ++ noun ++ ", ' " ++ key ++
      "'. Valid choices are: " ++
      (englishJoin "and" $ map fst l)
  m -> case lookup key m of
      Just x -> return x
      Nothing -> fail $
        "' " ++ key ++ "' is ambiguous. It could mean " ++
        (englishJoin "or" $ map fst m)

```

partialLookup is a helper function to look up a value in an associative array keyed on strings. The lookup function can match partial strings if they are not ambiguous. For example, given the keys "foo", "bar", and "baz", "f", "fo", or "foo" will map to the value for foo, "bar" will map to the value for bar, but "b", and "ba" will be ambiguous and result in an error.

```

englishJoin :: String -> [String] -> String
englishJoin _ [] = []
englishJoin _ [x] = x
englishJoin op [a,b] = concat [a," ",op," ",b]
englishJoin op l = concat $ englishJoin' l
  where englishJoin' [x] = [op," ",x]
        englishJoin' (x:xs) = x:", ":englishJoin' xs
        englishJoin' [] = undefined

```

englishJoin is a helper function to join a list of strings as they would be joined in the english language.

```

englishJoin "and" ["foo","bar","baz"] = "foo, bar, and baz"

```

```

readIntegral :: (Integral a, Monad b) => String -> b a
readIntegral s =
  case trim isSpace s of
    [] -> fail "readIntegral: expected an integer, found an empty string"
    ('-':s') -> negate `liftM` readIntegral' s'
    s' -> readIntegral' s'
  where
    readIntegral' cs = do
      digits <- mapM digitToInt' cs
      return $ foldl (\a b -> a*10+b) 0 digits
    digitToInt' c
      | isDigit c = return $ fromIntegral $ digitToInt c
      | otherwise = fail $ "readIntegral: " ++ [c] ++ " is not a digit"

```

readIntegral simply converts a string into an integer (base 10). It should work more or less like Prelude.read. The advantage it has over Prelude.read is that it doesn't drag in the whole lexer framework (and therefor leads to smaller binaries).

```
readIntegralMatrix :: (Integral a, Monad b) => String -> b [[a]]
readIntegralMatrix s = do
  when (s==[]) $ fail "empty string found where matrix expected"
  m <- mapM (mapM readIntegral)
    $ map (splitManyOn (=='/'))
    $ splitManyOn (==' ') s
  unless (and $ map ((==length (head m)).length) m)
    $ fail "matrix rows are of unequal size"
  return m
```

`readIntegralMatrix` converts a string in the format: “r1c1,r1c2,r1c3/r2c1,r2c2,r2c3” to a matrix. If the string cannot be parsed as an integral matrix the monad’s `fail` function is called with an appropriate error message.

```
showMatrix :: Show a => [[a]] -> String
showMatrix = concat . intersperse "/"
  . map (concat . intersperse ", " . map show)
```

`showMatrix` converts a matrix to a string in the format described above.

```
listToTuple :: [a] -> (a,a)
listToTuple [a,b] = (a,b)
listToTuple _ = error "Brianweb.Misc.listToTuple: not a two element list"
```

```
fromEitherM :: (Show a, Monad m) => Either a b -> m b
fromEitherM (Left a) = fail ("fromEitherM: " ++ show a)
fromEitherM (Right b) = return b

fromEither :: (Show a) => Either a b -> b
fromEither (Left a) = error ("fromEither: " ++ show a)
fromEither (Right b) = b
```

```
genericToEnum :: (Integral i, Enum a) => i -> a
genericToEnum = toEnum . fromIntegral

genericFromEnum :: (Integral i, Enum a) => a -> i
genericFromEnum = fromIntegral . fromEnum
```

10.3 Operators

10.3.1 Composition Operators

```
(.!) :: (b -> c) -> (a -> a' -> b) -> (a -> a' -> c)
(.!) = ((.) . (.)
```

```
(.!!) :: (b -> c) -> (a -> a' -> a'' -> b) -> (a -> a' -> a'' -> c)
(.!!) = ((.!) . (.)
```

```
(.!!!) :: (b → c) → (a → a' → a'' → a''' → b) → (a → a' → a'' → a''' → c)
(.!!!) = ((!!!).(.)
```

10.3.2 Misc Operators

```
(<=>) :: Ord a ⇒ a → a → Ordering
(<=>) = compare
```

Bibliography

- [Ada93] Stephen Adams. Functional pearls: Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.
- [Coh93] Henri Cohen. *A course in computational algebraic number theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
- [DFM] *Data.FiniteMap*. www.haskell.org/ghc/docs/6.2/html/libraries/base/Data.FiniteMap.html.
- [IBT] `org.ibex.util.balancedtree`. util.ibex.org/src/org/ibex/util/BalancedTree.java.
- [KLS04] Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press.
- [TW02] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, USA, 2002.