

Impressive Haskell

Otakar Smrž

Institute of Formal and Applied Linguistics
Charles University in Prague

January 27, 2006

Haskell's overview

```
qsort []      = []
qsort (x:xs) = qsort [ y | y <- xs, y < x ] ++ [x] ++
               qsort [ y | y <- xs, y >= x ]
```

purely functional expressions are functions
functions are first-class values
data structures are non-mutable

Haskell's overview

```
qsort []      = []
qsort (x:xs) = qsort [ y | y <- xs, y < x ] ++ [x] ++
               qsort [ y | y <- xs, y >= x ]
```

purely functional expressions are functions
functions are first-class values
data structures are non-mutable

non-strict = lazy evaluation by need
easy modeling of infinite or cyclic structures

Haskell's overview

```
qsort []      = []
qsort (x:xs) = qsort [ y | y <- xs, y < x ] ++ [x] ++
               qsort [ y | y <- xs, y >= x ]
```

purely functional expressions are functions
functions are first-class values
data structures are non-mutable

non-strict = lazy evaluation by need
easy modeling of infinite or cyclic structures

typing system values have distinct types
functions are just another type of type :)
polymorphism, type classes, inference

Haskell's overview

```
qsort []      = []  
qsort (x:xs) = qsort [ y | y <- xs, y < x ] ++ [x] ++  
               qsort [ y | y <- xs, y >= x ]
```

- purely functional** expressions are functions
functions are first-class values
data structures are non-mutable
- non-strict = lazy** evaluation by need
easy modeling of infinite or cyclic structures
- typing system** values have distinct types
functions are just another type of type :)
polymorphism, type classes, inference
- λ in the logo** direct connection to lambda calculus
+ natural syntactic notation

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Our interests

Introduction

Functions & Types

Type Classes

Monadic Parsing

D-S Languages

Natural Language

References

Functions with types

- everything in a pure functional language is a **function**
- functions operate on values of some (nontrivial) **type**
- **computation** is an **evaluation** of function
- **side effects** are due to evaluation on 'hidden input'

```
['h','e','l','p'] :: String  
"123"           :: [Char]  
123             :: Num a => a  
([],[])        :: ([a],[b])  
words          :: String -> [String]
```

Functions with types

- everything in a pure functional language is a **function**
- functions operate on values of some (nontrivial) **type**
- **computation** is an **evaluation** of function
- **side effects** are due to evaluation on 'hidden input'

```
['h','e','l','p'] :: String
"123"           :: [Char]
123             :: Num a => a
([],[])         :: ([a],[b])
words           :: String -> [String]

main  :: IO ()
main  = do putStr "You say: "; theline <- getLine
          putStr $ "Haskell says: Why not " ++
                (unwords.reverse.words) theline ++ "?"
```

Currying and the λ

- named after Haskell Curry, used first by M. Schönfinkel
- equating **multi-parameter functions** to a series of **one-parameter functions** applied in sequence
- notational variants of one equivalent thing in **λ -calculus**
- **new expressivity**: partial application, function composition

```
triple      :: a -> b -> c -> (a,b,c)
triple x y z = (x,y,z)
```

```
triple'    :: ( a -> b -> c -> (a,b,c) )
triple'    = \ x y z -> (x,y,z)
```

```
triple''   :: a -> ( b -> ( c -> (a,b,c) ) )
triple''   = \ x -> ( \ y -> ( \ z -> (x,y,z) ) )
```

Currying and the λ

- named after Haskell Curry, used first by M. Schönfinkel
- equating **multi-parameter functions** to a series of **one-parameter functions** applied in sequence
- notational variants of one equivalent thing in **λ -calculus**
- **new expressivity**: partial application, function composition

```
triple      :: a -> b -> c -> (a,b,c)
triple x y  = \ z -> (x,y,z)
```

```
triple'     :: a -> ( b -> c -> (a,b,c) )
triple' x   = \ y z -> (x,y,z)
```

```
triple''    :: a -> ( b -> ( c -> (a,b,c) ) )
triple'' x  = \ y -> ( \ z -> (x,y,z) )
```

Currying and the λ

- named after Haskell Curry, used first by M. Schönfinkel
- equating **multi-parameter functions** to a series of **one-parameter functions** applied in sequence
- notational variants of one equivalent thing in **λ -calculus**
- **new expressivity**: partial application, function composition

```
triple      :: a -> b -> c -> (a,b,c)
triple x    = \ y z -> (x,y,z)
```

```
triple'     :: a -> b -> ( c -> (a,b,c) )
triple' x y = \      z -> (x,y,z)
```

```
triple''    :: a -> ( b -> ( c -> (a,b,c) ) )
triple'' x y =                \ z -> (x,y,z)
```

Currying and the λ

- named after Haskell Curry, used first by M. Schönfinkel
- equating **multi-parameter functions** to a series of **one-parameter functions** applied in sequence
- notational variants of one equivalent thing in **λ -calculus**
- **new expressivity**: partial application, function composition

```
triple      :: a -> b -> c -> (a,b,c)
triple     = \ x y z -> (x,y,z)
```

```
triple'    :: a -> b -> c -> ( (a,b,c) )
triple' x y z =                (x,y,z)
```

```
triple''   :: a -> ( b -> ( c -> (a,b,c) ) )
triple'' x y z =                (x,y,z)
```

Endlessness

```
primes      :: [Int]
primes     = sieve [ 2 .. ]    -- the sieve of Eratosthenes

sieve      :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, y `rem` x /= 0 ]
```

Endlessness

```
primes      :: [Int]
primes      = sieve [ 2 .. ]    -- the sieve of Eratosthenes
```

```
sieve       :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, y `rem` x /= 0 ]
sieve []    = []
```

```
isPrimeIn   :: Int -> [Int] -> Bool
isPrimeIn x l = x `elemInc` sieve l
  where elemInc x (y:ys) | x > y      = elemInc x ys
                        | x == y      = True
                        | otherwise   = False
        elemInc _ []                = False
```

Endlessness

```
primes      :: [Int]
primes     = sieve [ 2 .. ]    -- the sieve of Eratosthenes
```

```
sieve      :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, y `rem` x /= 0 ]
sieve []    = []
```

```
isPrimeIn  :: Int -> [Int] -> Bool
isPrimeIn x l = x `elemInc` sieve l
  where elemInc x (y:ys) | x > y      = elemInc x ys
                        | x == y     = True
                        | otherwise  = False
        elemInc _ []                = False
```

Endlessness

```

primes      :: [Int]
primes     = sieve [ 2 .. ]    -- the sieve of Eratosthenes

sieve      :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, y `rem` x /= 0 ]
sieve []    = []

isPrime    :: Int -> Bool
isPrime x  = isPrimeIn x primes

isPrimeIn  :: Int -> [Int] -> Bool
isPrimeIn x l = x `elemInc` sieve l
  where elemInc x (y:ys) | x > y      = elemInc x ys
                        | x == y     = True
                        | otherwise  = False
                        elemInc _ []  = False

```

Higher-order functions

$$\begin{aligned}
 [2,3,5,7] &\equiv 2:3:5:7:[] \equiv (:) 2 \\
 &\quad ((:) 3 \\
 &\quad \quad ((:) 5 \\
 &\quad \quad \quad ((:) 7 [])))
 \end{aligned}$$

```

map'      :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
  
```

Higher-order functions

$$\begin{aligned}
 [2,3,5,7] &\equiv 2:3:5:7:[] \equiv (:) 2 \\
 &\quad ((:) 3 \\
 &\quad \quad ((:) 5 \\
 &\quad \quad \quad ((:) 7 [])))
 \end{aligned}$$

```

filter'      :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs) = if f x then x : filter' f xs
                  else filter' f xs
  
```

Higher-order functions

$$\begin{aligned}
 [2,3,5,7] &\equiv 2:3:5:7:[] \equiv (:) 2 \\
 &\quad ((:) 3 \\
 &\quad \quad ((:) 5 \\
 &\quad \quad \quad ((:) 7 [])))
 \end{aligned}$$

```

map'      :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
  
```

```

filter'   :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' f (x:xs) = if f x then x : filter' f xs
                  else filter' f xs
  
```

```

map      f l      = [ f x | x <- l ]
filter f l      = [ x | x <- l, f x ]
  
```

Further abstraction

$$\begin{aligned}
 [2,3,5,7] &\equiv 2:3:5:7:[] \equiv (:) 2 \\
 &\quad ((:) 3 \\
 &\quad \quad ((:) 5 \\
 &\quad \quad \quad ((:) 7 [])))
 \end{aligned}$$

```

map'      :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
  
```

Further abstraction

$$(2+3+5+7) \equiv 2+3+5+7+ 0 \equiv (+) 2$$

$$\quad \quad \quad ((+) 3$$

$$\quad \quad \quad \quad ((+) 5$$

$$\quad \quad \quad \quad \quad ((+) 7 \ 0)))$$

```
map'      :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
```

```
sum'      :: Num a => [a] -> a
sum' []   = 0
sum' (x:xs) = x : sum' xs
```


Further abstraction

$$c\ 2\ (c\ 3\ (c\ 5\ (c\ 7\ 0))) \equiv (c)\ 2$$

$$\quad\quad\quad (c)\ 3$$

$$\quad\quad\quad\quad (c)\ 5$$

$$\quad\quad\quad\quad\quad (c)\ 7\ z\)\)\)$$

```
map'      :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f ((:) x xs) = (:) (f x) (map' f xs)
```

```
sum'      :: Num a => [a] -> a
sum' []   = 0
sum' ((:) x xs) = (+) (x) (sum' xs)
```

```
foldr'    :: (a -> b -> b) -> b -> [a] -> b
foldr' c z [] = z
foldr' c z (x:xs) = c x (foldr' c z xs)
```

Functional composition

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

```
(.) f g x = f (g x)
```

-- useful concept for simplifications of code, intuitive interpretation

```
map' :: (a -> b) -> [a] -> [b]
```

```
map' _ [] = []
```

```
map' f ((:) x xs) = ((:) (f x)) (map' f xs)
```

```
sum' :: Num a => [a] -> a
```

```
sum' [] = 0
```

```
sum' ((:) x xs) = (+) (x) (sum' xs)
```

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr' c z [] = z
```

```
foldr' c z (x:xs) = c x (foldr' c z xs)
```

Functional composition

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

```
(.) f g x = f (g x)
```

-- useful concept for simplifications of code, intuitive interpretation

```
map' :: (a -> b) -> [a] -> [b]
```

```
map' _ [] = []
```

```
map' f ((:) x xs) = ((:).f) (x) (map' f xs)
```

```
sum' :: Num a => [a] -> a
```

```
sum' [] = 0
```

```
sum' ((:) x xs) = (+) (x) (sum' xs)
```

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr' c z [] = z
```

```
foldr' c z (x:xs) = (c) (x) (foldr' c z xs)
```

Functional composition

```
(.) :: (a -> b) -> (c -> a) -> c -> b
```

```
(.) f g x = f (g x)
```

-- useful concept for simplifications of code, intuitive interpretation

```
map' :: (a -> b) -> [a] -> [b]
```

```
map' f = foldr' ((:).f) []
```

```
sum' :: Num a => [a] -> a
```

```
sum' = foldr' (+) 0
```

```
foldr' :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr' c z [] = z
```

```
foldr' c z (x:xs) = (c) (x) (foldr' c z xs)
```

Data types

- data types are **sets of constant values** expressing structure
- values brought into existence through **constructor functions**

```
data Case = Nominative | Genitive | Accusative
```

```
data Info = Info String Case
```

Data types

- data types are **sets of constant values** expressing structure
- values brought into existence through **constructor functions**

```
data Case = Nominative | Genitive | Accusative
```

```
data Info = Info { lemma :: String, inCase :: Case }
```

Data types

- data types are **sets of constant values** expressing structure
- values brought into existence through **constructor functions**

```
data Case = Nominative | Genitive | Accusative
```

```
data Info = Info { lemma :: String, inCase :: Case }
```

```
data List a = Null | Cons a (List a)
```

```
Cons "List" (Cons "of" (Cons "Strings" Null))
```

Data types

- data types are **sets of constant values** expressing structure
- values brought into existence through **constructor functions**

```
data Case = Nominative | Genitive | Accusative
```

```
data Info = Info { lemma :: String, inCase :: Case }
```

```
data List a = Null | Cons a (List a)
```

```
Cons "List" (Cons "of" (Cons "Strings" Null))
```

```
data Tree a b = Leaf a | Fork b [ Tree a b ]
```

```
Fork "root" [ Leaf 1,  
              Fork "fork" [ Leaf 2,  
                           Leaf 3 ],  
              Leaf 4,  
              Leaf 5 ]
```

Type classes

- restrict **type polymorphism**, express rich **type relations**
- provide **function overloading** and promote **modularity**

```
data UPoint = UPoint Int    -- unique identifier ~ Unicode
```

```
class Encoding e where  
  encode    :: e -> [UPoint] -> [Char]  
  decode    :: e -> [Char] -> [UPoint]
```

Type classes

- restrict **type polymorphism**, express rich **type relations**
- provide **function overloading** and promote **modularity**

```
module Encode (Encoding, UPoint, encode, decode) where
```

```
data UPoint = UPoint Int    -- unique identifier ~ Unicode
```

```
class Encoding e where
```

```
    encode    :: e -> [UPoint] -> [Char]
```

```
    decode    :: e -> [Char] -> [UPoint]
```

```
    encode _ = map (toEnum . fromEnum)
```

```
    decode _ = map (toEnum . fromEnum)
```

```
instance Enum UPoint where
```

```
    fromEnum (UPoint x) = x
```

```
    toEnum      = UPoint
```

Showing

```
show $ BFork (BLeaf 'a') (BFork (BLeaf 'x') (BLeaf 'y'))  
  → "BFork (BLeaf 'a') (BFork (BLeaf 'x') (BLeaf 'y'))"
```

```
class Show a where
```

```
  show      :: a -> String
```

```
  showsPrec :: Int -> a -> String -> String
```

```
  show x      = showsPrec 0 x ""
```

```
  showsPrec _ x s = show x ++ s
```

```
data BTree a = BFork (BTree a) (BTree a) | BLeaf a  
  deriving Show
```

Showing

```
show $ BFork (BLeaf 'a') (BFork (BLeaf 'x') (BLeaf 'y'))  
  → "<'a'|<'x'|'y'>>"
```

```
class Show a where
```

```
  show           :: a -> String
```

```
  showsPrec     :: Int -> a -> String -> String
```

```
data BTree a = BFork (BTree a) (BTree a) | BLeaf a
```

```
instance Show a => Show (BTree a) where
```

```
  show = showBTree
```

```
showBTree           :: Show a => BTree a -> String
```

```
showBTree (BLeaf x) = show x
```

```
showBTree (BFork l r) = "<" ++ showBTree l ++ "|"  
                        ++ showBTree r ++ ">"
```

Monads

- complex types 'providing' values of some other type but 'encapsulating' more information, with functions + laws
- allow flexible structuring of the computation, no magic :)

```
class Monad m where
  (>>=)      :: m a -> (a -> m b) -> m b
  (>>)       :: m a -> m b -> m b
  return     :: a -> m a
  fail       :: String -> m a
```

Monads

- complex types 'providing' values of some other type but 'encapsulating' more information, with functions + laws
- allow flexible structuring of the computation, no magic :)

```
class Monad m where
  (>>=)      :: m a -> (a -> m b) -> m b
  (>>)       :: m a -> m b -> m b
  return     :: a -> m a
  fail       :: String -> m a

  (>>) x y = (>>=) x (\ _ -> y)
```

```
instance Monad [] where
  (>>=) x y = concat (map y x)
  return x  = [x]
  fail     = []
```

Do syntax

- easy notation to use with monadic types, directly translated to ($\gg=$), return and the like, and to λ -expressions

```
[ (x,y) | x <- [ 1 .. 5 ], y <- [ 5, 3 .. -3 ], x /= y ]
```

```
[(1,5), (1,3), (1,-1), (1,-3), (2,5), (2,3), (2,1), (2,-1), (2,-3),  
(3,5), (3,1), (3,-1), (3,-3), (4,5), (4,3), (4,1), (4,-1), (4,-3),  
(5,3), (5,1), (5,-1), (5,-3)]
```

Do syntax

- easy notation to use with monadic types, directly translated to ($\gg=$), return and the like, and to λ -expressions

```
[ (x,y) | x <- [ 1 .. 5 ], y <- [ 5, 3 .. -3 ], x /= y ]
```

```
[(1,5), (1,3), (1,-1), (1,-3), (2,5), (2,3), (2,1), (2,-1), (2,-3),  
(3,5), (3,1), (3,-1), (3,-3), (4,5), (4,3), (4,1), (4,-1), (4,-3),  
(5,3), (5,1), (5,-1), (5,-3)]
```

```
do x <- [ 1 .. 5 ]  
   y <- [ 5, 3 .. -3 ]  
   True <- return (x /= y)  
   return (x,y)
```

Do syntax

- easy notation to use with monadic types, directly translated to ($\gg=$), return and the like, and to λ -expressions

```
[ (x,y) | x <- [ 1 .. 5 ], y <- [ 5, 3 .. -3 ], x /= y ]
```

```
[(1,5), (1,3), (1,-1), (1,-3), (2,5), (2,3), (2,1), (2,-1), (2,-3),
 (3,5), (3,1), (3,-1), (3,-3), (4,5), (4,3), (4,1), (4,-1), (4,-3),
 (5,3), (5,1), (5,-1), (5,-3)]
```

```
do x <- [ 1 .. 5 ]           [ 1 .. 5 ]      >>= \ x ->
  y <- [ 5, 3 .. -3 ]       [ 5, 3 .. -3 ]  >>= \ y ->
  True <- return (x /= y)   return (x /= y) >>= \ r ->
  return (x,y)              case r of
                             True -> return (x,y)
                             _     -> fail "i.e. []"
```

Reading

```
read      :: String -> a    -- defined in the Prelude module
```

```
class Read a where
```

```
  readsPrec      :: Int -> String -> [(a,String)]
```

```
readsBTree      :: Read a => String -> [(BTree a,String)]
```

```
readsBTree ('<':s) = [ (BFork l r, u) |  
                      (l, '|':t) <- readsBTree s,  
                      (r, '>':u) <- readsBTree t ]
```

```
readsBTree s      = [ (BLeaf x, t)   | (x, t) <- reads s ]
```

Reading

```
read      :: String -> a    -- defined in the Prelude module
```

```
class Read a where
```

```
    readsPrec      :: Int -> String -> [(a,String)]
```

```
readsBTree  :: Read a => String -> [(BTree a,String)]
```

```
readsBTree s = [ (BFork l r, x) | (" $<$ ", t) <- lex s,  
                                (l, u) <- readsBTree t,  
                                (" $|$ ", v) <- lex u,  
                                (r, w) <- readsBTree v,  
                                (" $>$ ", x) <- lex w ]
```

```
++
```

```
[ (BLeaf x, t)   | (x, t) <- reads s ]
```

```
instance Read a => Read (BTree a) where
```

```
    readsPrec _ = readsBTree    -- implies the definition of read
```

The Parser

```
type Parser a = String -> [(a,String)]
```

The Parser

```
data Parser a = Parser (String -> [(a,String)])
```

```
parse          :: Parser a -> String -> [(a,String)]
```

```
parse (Parser p) = p
```

The Parser

```
data Parser a = Parser (String -> [(a,String)])
```

```
parse          :: Parser a -> String -> [(a,String)]
```

```
parse (Parser p) = p
```

```
itemParser     :: Parser Char
```

```
itemParser     = Parser ( \ s -> case s of c:r -> [(c,r)]  
                        ""  -> [] )
```

```
consParser     :: a -> Parser a
```

```
consParser x   = Parser ( \ s -> [(x,s)] )
```

The Parser

```
data Parser a = Parser (String -> [(a,String)])
```

```
parse          :: Parser a -> String -> [(a,String)]
```

```
parse (Parser p) = p
```

```
itemParser     :: Parser Char
```

```
itemParser     = Parser ( \ s -> case s of c:r -> [(c,r)]  
                        ""   -> [] )
```

```
consParser     :: a -> Parser a
```

```
consParser x   = Parser ( \ s -> [(x,s)] )
```

```
parse itemParser "this is to be parsed"
```

```
  → [('t',"his is to be parsed")]
```

```
parse (consParser (+)) "this will remain unchanged"
```

```
  → [(+),"this will remain unchanged"]
```

Parser monad

```
joinParser      :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                          parse (q r) t | (r,t) <- parse p s ] )
```

Parser monad

```
joinParser      :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                          parse (q r) t | (r,t) <- parse p s ] )
```

```
instance Monad Parser where
```

```
    (>>=) = joinParser
```

```
    return = consParser
```

```
pairParser :: Parser (Char,Char)
```

```
pairParser = do x <- itemParser
                y <- itemParser
                return (x,y)
```

Parser monad

```
joinParser      :: Parser a -> (a -> Parser b) -> Parser b
joinParser p q = Parser ( \ s -> concat [
                          parse (q r) t | (r,t) <- parse p s ] )
```

```
instance Monad Parser where
```

```
    (>>=) = joinParser
```

```
    return = consParser
```

```
pairParser :: Parser (Char,Char)
```

```
pairParser = do x <- itemParser
                y <- itemParser
                return (x,y)
```

```
parse pairParser "this is to be parsed"
```

```
  → [(( 't', 'h' ), "is is to be parsed")]
```

```
parse (do pairParser; pairParser) "this is to be parsed"
```

```
  → [(( 'i', 's' ), " is to be parsed")]
```

Parsing XML

```
data XML = Element ElementName [Attribute] [XML] |  
          Another String
```

```
type ElementName = String
```

```
type Attribute = (String,String)
```

```
xmlelt :: Parser XML
```

```
xmlelt = do (e,as) <- opening  
          xs <- many (xmlelt +++ pCDATA)  
          closing e  
          return (Element e as xs)
```

```
closing, identify :: ElementName -> Parser ()
```

```
closing e = do char '<'; char '/'; identify e; char '>'  
           return ()
```

```
identify e = case e of c:r -> do char c; identify r  
                "" -> return ()
```

HaXml

```

data ElemXML = ElemXML ElementName [Attribute] [Content]
data Content = CElem ElemXML | CText String

type CFilter = Content -> [Content]

scene :: CFilter
scene = tag "SCENE"
      ?> (h3 [txt 'o' children 'o' tag "TITLE" 'o' children])
        'union'
        (children 'without' (tag "TITLE"))
      :> keep

deleteSpeechOf      :: String -> CFilter
deleteSpeechOf spkr = foldXml
  (tag "SPEECH" </
    (tag "SPEAKER" /> txt 'with' pcdDataFilter spkr)
    ?> none :> keep)

```

WASH

```
module Main where

import Prelude hiding (map, span, head, div)
import CGI

main = run mainCGI

mainCGI = counter 0

counter n =
  standardQuery "Counter" $ p_T $
  do text_S "Current counter value "
      text_S (show n)
      br_S empty
      submit0 (counter (n + 1)) (fieldVALUE "Increment")
      submit0 (counter (n - 1)) (fieldVALUE "Decrement")
```

Functional Morphology

- library for **high-level** declaration of morphological systems hiding any **low-level** implementation issues from the users
- reduced to defining **grammatical categories** and **inflection paradigms** and **word classes** and listing **lexical items**

```
data Case = Nominative | Genitive | Accusative | Ablative |  
           Dative | Vocative
```

```
data Number = Singular | Plural
```

```
data NounInfl = NounInfl Number Case
```

Functional Morphology

```
ourParadigm :: String -> NounInfl -> String
ourParadigm rosa (NounInfl n c) =
  let rosae = rosa ++ "e"
      rosis = init rosa ++ "is"
  in case n of
    Singular -> case c of
      Accusative -> rosa ++ "m"
      Genitive    -> rosae
      Dative      -> rosae
      _           -> rosa
    Plural -> case c of
      Nominative -> rosae
      Vocative   -> rosae
      Accusative -> rosa ++ "s"
      Genitive   -> rosa ++ "rum"
      _         -> rosis
```

Programming as writing

:)

References

- The Haskell School of Expression
- How to Declare an Imperative
- Pure Functional Parsing
- Using Types to Parse Natural Language
- Functional Pearl: The Zipper
- Advanced Programming and the DSLs
- Documentations to Haddock, lhs2TeX
- Higher-Order Perl, Beamer
- Yet Another Haskell Tutorial

www.haskell.org