Functional Programming, FPR
3rd - 7th October 2011
ASSIGNMENT

Tim Pizey
Wellcome Trust Centre for Human Genetics
Tim.Pizey@well.ox.ac.uk

Page count 32

April 10, 2014

[Page intentionally left blank]

1 Part I

1.0.1 Setup

This section describes how to set the project up and what output to expect.

Load this whole file into the Glasgow Haskell Compiler Interactive interface (GHCi).

The following warnings are expected:

```
FPR.lhs:517:3: Warning: Defined but not used: ‘badList’

FPR.lhs:517:3:
    Warning: Top-level binding with no type signature:
               badList :: forall a. Sequ a -> [a]

FPR.lhs:540:3: Warning: Defined but not used: ‘nonTerminating’
```

this is due to *badList* being an example of bad style and *nonterminating* not terminating, as the names suggest.

Warning level is set to *all*:

{-# OPTIONS -Wall #-}

1.0.2 Imported modules and copied functions

The following Prelude *List* functions are hidden as we are going to use them with *Sequ*s.

```
module FPR (main) where
import Prelude hiding ((++), last, reverse, length,
  map, concat, concatMap)
import Control.Applicative
import Control.Monad
```

The hiding of these functions means that to use them, for example in tests, they have to be copied directly into this module. We will still need to append lists, so we will use *+++*.

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

  -- Data.List.last
listLast          :: [a] -> a
listLast [x]      = x
listLast (_ : xs) = listLast xs
listLast []       = error "empty list in listLast"

  -- Data.List.length
listLength :: [a] -> Int
listLength []       = 0
listLength (_ : xs) = 1 + listLength xs

  -- Data.List.map
listMap :: (a -> b) -> [a] -> [b]
listMap _ [] = []
```

```
listMap f (x : xs) = f x : listMap f xs
  -- Data.List.concat
  -- Concatenate a list of lists.
listConcat :: [[a]] → [a]
listConcat = foldr (++) []
  -- Data.List.concatMap
  -- Map a function over a list and concatenate the results.
listConcatMap   :: (a → [b]) → [a] → [b]
listConcatMap f = foldr ((++) ∘ f) []
  -- Data.List.nub
nub :: Eq a ⇒ [a] → [a]
nub l = nub' l []
  where
    nub' [] _          = []
    nub' (x : xs) ls
      | x ∈ ls         = nub' xs ls
      | otherwise      = x : nub' xs (x : ls)
```

1.0.3 Testing

Rather than pasting the results of functions into the body of the text equality assertions are used. These are run when the file is compiled and executed (a much lighter version of [HUnit]).

```
type Assertion = Bool
assertEqual :: (Eq a) ⇒ a → a → Bool
assertEqual expected actual = expected ≡ actual
tests :: [Assertion]
tests = [test1, test2, test3, test4, test5, test6, test7, test8,
  test9, test10, test11, test12, test13, test14, test15,
  test16, test17, test18, test19, test20, test21, test22,
  test23, test24, test25, test26, test27, test28, test29,
  test30, test31, test32, test33, test34, test35, test36,
  test37, test38, test39, test40, test41, test42]
allWell :: Bool
allWell = and tests
main :: IO ()
main = if allWell then do output ≫ putStr ("OK\n")
  else putStr ((show tests) ++ "\n")
```

To test all functions execute *main*:

```
*FPR> main
OK
```

1.1 Leaf Trees

We want the list of the possible *Leaf Trees* whose leaves in left to right order are the list of the first five primes. A *Leaf Tree* has values (labels) only at leaves, here called *Sequ*:

```
data Sequ a = Empty | Single a | Cat (Sequ a) (Sequ a)
  deriving (Show, Eq, Ord)
```

1.1.1 The derivation of all *Sequ*

In the following the *Cat* constructor is represented with parentheses and the *Single* constructor is ommitted.

We can describe the *Sequ*s in terms of the most simple and the process for adding nodes to build up larger *Sequ*. We know that there is only one order-preserving *Sequ* for the empty list, a singleton list and a two element list. For *Sequ* with more elements the new element can be added by being appended to the whole *Sequ* or inside any of the rightmost *Sequ*.

To generate all the possible *Sequ* with leaves in the same order as a *List* :

```
leafTrees :: [a] -> [Sequ a]
leafTrees a = leafTrees' a [Empty]

leafTrees' :: [a] -> [Sequ a] -> [Sequ a]
leafTrees' [] soFar = soFar
leafTrees' (x : xs) soFar =
    leafTrees' xs (listConcatMap (addLeaf x) soFar)

addLeaf :: a -> Sequ a -> [Sequ a]
addLeaf x Empty = [Single x]
addLeaf y (Single x) = [Cat (Single x) (Single y)]
addLeaf x (Cat (l) (r)) =
    [Cat (Cat (l) (r)) (Single x)]
     +++
    (listMap (Cat (l)) (addLeaf x r))

firstFivePrimeLeafTrees :: [Sequ Integer]
firstFivePrimeLeafTrees = leafTrees [2, 3, 5, 7, 11]

test1 :: Assertion
test1 = assertEqual
  [Cat (Cat (Cat (Cat (Single 2) (Single 3)) (Single 5)) (Single 7)) (Single 11),
    Cat (Cat (Cat (Single 2) (Single 3)) (Single 5)) (Cat (Single 7) (Single 11)),
    Cat (Cat (Cat (Single 2) (Single 3)) (Cat (Single 5) (Single 7))) (Single 11),
    Cat (Cat (Single 2) (Single 3)) (Cat (Cat (Single 5) (Single 7)) (Single 11)),
    Cat (Cat (Single 2) (Single 3)) (Cat (Single 5) (Cat (Single 7) (Single 11))),
    Cat (Cat (Cat (Single 2) (Cat (Single 3) (Single 5))) (Single 7)) (Single 11),
    Cat (Cat (Single 2) (Cat (Single 3) (Single 5))) (Cat (Single 7) (Single 11)),
    Cat (Cat (Single 2) (Cat (Cat (Single 3) (Single 5)) (Single 7))) (Single 11),
    Cat (Single 2) (Cat (Cat (Cat (Single 3) (Single 5)) (Single 7)) (Single 11)),
    Cat (Single 2) (Cat (Cat (Single 3) (Single 5)) (Cat (Single 7) (Single 11))),
    Cat (Cat (Single 2) (Cat (Single 3) (Cat (Single 5) (Single 7)))) (Single 11),
    Cat (Single 2) (Cat (Cat (Single 3) (Cat (Single 5) (Single 7))) (Single 11)),
```

```
    Cat (Single 2) (Cat (Single 3) (Cat (Cat (Single 5) (Single 7)) (Single 11))),
    Cat (Single 2) (Cat (Single 3) (Cat (Single 5) (Cat (Single 7) (Single 11))))
  ]
  firstFivePrimeLeafTrees
```

1.1.2 *Catalan Number* tests

We know that the number of possible binary trees from with *n + 1* leaves is the *Catalan Number* index *n* [Binary Trees]

```
test2 :: Assertion
test2 = assertEqual 14 (listLength firstFivePrimeLeafTrees)
  -- from Math-Combinat-Numbers
binomial :: Integral a ⇒ a → a → Integer
binomial n k
  | k > n = 0
  | k < 0 = 0
  | k > (n `div` 2) = binomial n (n − k)
  | otherwise = (product [n′ − k′ + 1 .. n′]) `div` (product [1 .. k′])
  where
    k′ = fromIntegral k
    n′ = fromIntegral n
  -- from Math-Combinat-Trees-Binary
catalan :: Int → Integer
catalan n = binomial (2 ∗ n) n `div` (1 + fromIntegral n)
  -- Test equality of members of a pair
assertEqualIntegralTuple :: (Integral a, Integral b) ⇒ (a, b) → Bool
assertEqualIntegralTuple (x, y) =
  assertEqual x (fromIntegral y)
  -- Count possible Sequ
sequCounts :: Int → [(Integer, Int)]
sequCounts n = [(catalan (x − 1),
  (listLength (leafTrees [1 .. x]))) | x ← [1 .. n]]
```

The following starts to take a long time quite quickly!

```
test3 :: Assertion
test3 = and $ listMap assertEqualIntegralTuple
  $ sequCounts 11
```

1.1.3 Uniqueness test

A buggy implementation might generate duplicates, so we guard against that with a test.

```
test4 :: Assertion
test4 = assertEqual firstFivePrimeLeafTrees
  (nub firstFivePrimeLeafTrees)
```

1.2 Leaf Tree Concatenation

Two leaf trees are appended when the lists of their leaves are concatenated.

```
(++) :: Sequ a -> Sequ a -> Sequ a
(++) Empty y = y
(++) x Empty = x
(++) x y = Cat (x) (y)
```

The type signature tells us that the function takes two arguments of type *Sequ a* that is a *Sequ* of any type (*Sequ* is a parametric data type). The function yields a *Sequ* value.

```
test5 :: Assertion
test5 = assertEqual
  (Cat (Cat (Single '1') (Single '2')) (Cat (Single '3') (Single '4')))
  (Cat (Single '1') (Single '2') ++ Cat (Single '3') (Single '4'))

test6 :: Assertion
test6 = assertEqual (['1','2'] ++ ['3','4'])
  (list ((Cat (Single '1') (Single '2') ++ Cat (Single '3') (Single '4'))))
```

1.2.1 Curried *Sequ*

The signature also tells us that the function could be curried into a single argument function by supplying only one *Sequ* argument, though the resultant function is no longer polymorphic, its type being determined by the type of the supplied argument.

```
s1 :: Sequ Char
s1 = (Single 'x')
s2 :: Sequ Char
s2 = (Single 'y')

append' :: Sequ Char -> Sequ Char
append' = ((++) s1)

test7 :: Assertion
test7 = assertEqual ((++) s1 s2)
  (append' s2)
```

1.3 *Sequ* Design Pattern

The *Sequ* Design Pattern can be stated by analogy to the *List* Design Pattern.

1.3.1 *List* Design Pattern

List Design Patttern (List Consumer) [FPR 2011]

Task : Define a function of type f :: [P] → S

Step 1 : Solve the problem for the empty list
```
f [] = ...
```

Step 2: Assume that you already have the solution for "xs" ; extend the intermediate solution for x:xs

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

1.3.2 *Sequ* Design Pattern

In the same fashion we can define a design pattern for functions which consume an element of type *Sequ* T.

***Sequ* Design Patttern (*Sequ* Consumer)**

Task : Define a function of type consume :: *Sequ* → S

Step 1 : Solve the problem for the empty *Sequ* (Empty)
```
consume Empty = ...
```

Step 2 : Solve the problem for the leaf *Sequ* (Single)
```
consume Single x = ...
```

Step 3: Assume that you already have the solution for *Sequ*
(consume s); extend the intermediate solution for
Cat (Sequ l) (Sequ r)
by supplying the two argument function to conjoin the left and right branches (*conj*)

```
consume(Cat (l) (r)) = conj (consume l) (consume r )
```

1.3.3 A pretty printer for *Sequ*

For example a pretty printer for *Sequ*

pprint :: (*Show a*) ⇒ *Sequ a* → *String*
pprint Empty = `"E"`

```
pprint (Single x) = show x
pprint (Cat (l) (r)) = "(" +++ pprint l +++ ","
  +++ pprint r +++ ")"
test8 :: Assertion
test8 = assertEqual
  [
    "((((2,3),5),7),11)",
    "(((2,3),5),(7,11))",
    "(((2,3),(5,7)),11)",
    "((2,3),((5,7),11))",
    "((2,3),(5,(7,11)))",
    "(((2,(3,5)),7),11)",
    "((2,(3,5)),(7,11))",
    "((2,((3,5),7)),11)",
    "(2,(((3,5),7),11))",
    "(2,((3,5),(7,11)))",
    "((2,(3,(5,7))),11)",
    "(2,((3,(5,7)),11))",
    "(2,(3,((5,7),11)))",
    "(2,(3,(5,(7,11))))"]
  (listMap pprint firstFivePrimeLeafTrees)
```

[Bird 1998] page 180 gives us that there are five ways of bracketing four values.

```
test9 :: Assertion
test9 = assertEqual 5 (listLength (listMap pprint (leafTrees [1,2,3,4 :: Int])))
```

1.3.4 Design considerations for *Sequ* Producers

The considerations for a *Sequ* Producer are:
What patterns, if any, result in *Empty*?
What patterns result in a *Single*?
What patterns, if any, result in *Cat*?
Should a *Cat* be extended to the right or left?
When a *Cat* results does the *Sequ* need to be rebalanced?

1.3.5 Reverse *Sequ*

```
reverse :: Sequ a → Sequ a
reverse Empty    = Empty
reverse (Single a) = (Single a)
reverse (Cat l r) = (Cat (reverse r) (reverse l))
test10 :: Assertion
test10 = assertEqual
  (Cat (Single 'b') (Single 'a'))
  (reverse (Cat (Single 'a') (Single 'b')))
```

1.3.6 Last *Sequ*

$last :: Sequ\ a \to a$
$last\ Empty \quad = \bot$
$last\ (Single\ a) = a$
$last\ (Cat\ \_\ r) = last\ r$

$test11 :: Assertion$
$test11 = assertEqual$ `'g'` $(last\ (sequ$ `"Dog"`$))$

1.3.7 Maybe Last *Sequ*

```
(last (Cat (Single '0') Empty))
```

results in

```
*** Exception:  Prelude.undefined
```

As the return of a *Sequ* with a final *Empty* is *undefined* this would be better as a *Maybe* type:

$maybeLast :: Sequ\ a \to Maybe\ a$
$maybeLast\ Empty \quad = Nothing$
$maybeLast\ (Single\ x) = Just\ x$
$maybeLast\ (Cat\ \_\ r) = maybeLast\ (r)$

$test12 :: Assertion$
$test12 = assertEqual$
$Nothing$
$(maybeLast\ (Cat\ (Single$ `'0'`$)\ Empty))$

1.3.8 Length *Sequ*

$length :: Sequ\ a \to Integer$
$length\ Empty = 0$
$length\ (Single\ \_) = 1$
$length\ (Cat\ l\ r) = (length\ l) + (length\ r)$

$test13 :: Assertion$
$test13 = assertEqual\ 2\ (length\ (Cat\ (Cat\ (Single$ `'a'`$)\ (Single$ `'b'`$))\ Empty))$

A more general type for *length* would be

```
length :: (Integral n) => Sequ a -> n
```

1.4 Higher order functions

1.4.1 Map *Sequ*

$map :: (a \to b) \to (Sequ\ a \to Sequ\ b)$
$map\ \_\ Empty \quad = Empty$
$map\ f\ (Single\ a) = (Single\ (f\ a))$
$map\ f\ (Cat\ l\ r) = (Cat\ (map\ f\ l)\ (map\ f\ r))$

```
test14 :: Assertion
test14 = assertEqual (Cat (Single (2 :: Int)) (Single (3 :: Int)))
  (map ((+) 1) (Cat (Single (1 :: Int)) (Single (2 :: Int))))
```

1.4.2 Concat *Sequ*

```
concat :: Sequ (Sequ a) → Sequ a
concat Empty      = Empty
concat (Single a) = a
concat (Cat l r)  = (concat l) ++ (concat r)
test15 :: Assertion
test15 = assertEqual
  (Cat (Cat (Single '1') (Single '2')) (Cat (Single '3') (Single '4')))
  (concat
    (Cat (Single (Cat (Single '1') (Single '2')))
    (Single (Cat (Single '3') (Single '4'))))
  )
```

1.4.3 ConcatMap *Sequ*

```
concatMap :: (a → Sequ b) → (Sequ a → Sequ b)
concatMap _ Empty      = Empty
concatMap f (Single a) = f a
concatMap f (Cat l r)  = (concatMap f l) ++ (concatMap f r)
test16 :: Assertion
test16 = assertEqual
  (Cat (Cat (Single 'a') (Single 'b')) (Cat (Single 'c') (Single 'd')))
  (concatMap (sequ) (Cat (Single "ab") (Single "cd")))
```

1.4.4 *Sequ* as Functor

Sequ can be made an instance of *Functor* [Classes].

```
instance Functor Sequ where
  fmap _ Empty      = Empty
  fmap f (Single x) = Single (f x)
  fmap f (Cat l r)  = Cat (fmap f l) (fmap f r)
test17 :: Assertion
test17 = assertEqual (Cat (Single "f-a") (Single "f-b"))
  (fmap ((+++) "f-") (Cat (Single "a") (Single "b")))
```

1.4.5 *Sequ* as Applicative

```
instance Applicative Sequ where
  pure = return
  (<*>) = ap
test18 :: Assertion
```

```
test18 = assertEqual (Cat (Single "f-a") (Single "f-b"))
  (pure ((+++) "f-") <*> (Cat (Single "a") (Single "b")))
```

1.4.6 *Sequ* as Monad

```
instance Monad Sequ where
  return = Single
  Empty   >>= _ = Empty
  Single x >>= f = f x
  Cat l r  >>= f = Cat (l >>= f) (r >>= f)
  fail _ = Empty
test19 :: Assertion
test19 = assertEqual (Cat (Single "f-a") (Single "f-b"))
  ((Cat (Single "a") (Single "b")) >>= λx → return ("f-" +++ x))
```

1.5 Using the *Sequ* Design Pattern

1.5.1 *Sequ* Design Pattern as a higher order function

[Applogies for re-implementing all six functions, this was done for practice, use later on and symmetry. If this was inappropriate please consider *lengthC* and *concatMapC* as my submission for this section.]

The *List* Consumer Design Pattern can be written as

```
consumeList :: (a → b) → (b → c → c) → c → [a] → c
consumeList _ _ deflt [] = deflt
consumeList f conj deflt (x : xs) = conj (f x) (consumeList f conj deflt xs)
test20 :: Assertion
test20 = assertEqual (9 :: Int)
  (consumeList (+1) (+) (0) [1, 2, 3 :: Int])
test21 :: Assertion
test21 = assertEqual "123" (consumeList show (+++) "" [1, 2, 3 :: Int])
test22 :: Assertion
test22 = assertEqual 3
  (consumeList (\_ → 1) (+) (0 :: Int) ['a', 'b', 'c'])
```

The *Sequ* Consumer Design Pattern can similarly be written as follows (note that the default, intermediate and resultant types are the same)

```
consume :: (a → b) → (b → b → b) → b → Sequ a → b
consume _ _ deflt Empty        = deflt
consume f _ _ (Single x)       = (f x)
consume f conj deflt (Cat l r) = conj (consume f conj deflt l)
                                      (consume f conj deflt r)
test23 :: Assertion
test23 = assertEqual (9 :: Int)
  (consume (+1) (+) (0) (Cat (Cat (Single 1) (Single 2)) (Single 3)))
```

```
test24 :: Assertion
test24 = assertEqual "123"
  (consume show (+++) "" (Cat (Cat (Single (1 :: Int)) (Single 2)) (Single 3)))
```

1.5.2 Functions using *consume*

Reverse *Sequ*

```
reverseC :: Sequ a → Sequ a
reverseC = consume rewrap swap Empty where
  rewrap :: a → Sequ a
  rewrap x = (Single x)
  swap :: Sequ a → Sequ a → Sequ a
  swap l r = Cat r l

test25 :: Assertion
test25 = assertEqual
  (reverse (Cat (Single 'a') (Single 'b')))
  (reverseC (Cat (Single 'a') (Single 'b')))
```

Last *Sequ*

```
lastC :: Sequ a → a
lastC = consume id conj ⊥ where
  conj :: a → a → a
  conj _ r = r

test26 :: Assertion
test26 = assertEqual
  (last (sequ "Dog"))
  (lastC (sequ "Dog"))
maybeLastC :: Sequ a → Maybe a
maybeLastC = consume f conj Nothing where
  f :: a → Maybe a
  f x = Just x
  conj :: Maybe a → Maybe a → Maybe a
  conj _ r = r

test27 :: Assertion
test27 = assertEqual
  (maybeLast (Cat (Single '0') Empty))
  (maybeLastC (Cat (Single '0') Empty))
```

Length *Sequ*

```
lengthC :: Sequ a → Integer
lengthC = consume (\_ → 1) (+) 0

test28 :: Assertion
test28 = assertEqual
  (length (Cat (Cat (Single 'a') (Single 'b')) Empty))
  (lengthC (Cat (Cat (Single 'a') (Single 'b')) Empty))
```

1.5.3 Higher Order Functions using *consume*

Map *Sequ*

```
mapC :: (a → b) → (Sequ a → Sequ b)
mapC f = consume (fwrap f) Cat Empty where
  fwrap :: (a → b) → a → Sequ b
  fwrap fn x = (Single $ fn x)

test29 :: Assertion
test29 = assertEqual
  (map ((+) 1) (Cat (Single (1 :: Int)) (Single (2 :: Int))))
  (mapC ((+) 1) (Cat (Single (1 :: Int)) (Single (2 :: Int))))
```

Concat *Sequ*

```
concatC :: Sequ (Sequ a) → Sequ a
concatC = consume id (++) Empty

test30 :: Assertion
test30 = assertEqual
  (concat
    (Cat (Single (Cat (Single '1') (Single '2')))
    (Single (Cat (Single '3') (Single '4'))))
  )
  (concatC
    (Cat (Single (Cat (Single '1') (Single '2')))
    (Single (Cat (Single '3') (Single '4'))))
  )
```

ConcatMap *Sequ*

```
concatMapC :: (a → Sequ b) → (Sequ a → Sequ b)
concatMapC f = consume f (++) Empty

test31 :: Assertion
test31 = assertEqual
  (concatMap (sequ) (Cat (Single "ab") (Single "cd")))
  (concatMapC (sequ) (Cat (Single "ab") (Single "cd")))
```

1.6 *Sequ* to List

```
list :: Sequ a → [a]
list Empty = []
list (Single x) = [x]
list (Cat (l) (r)) = (list l) + + + (list r)

test32 :: Assertion
test32 = assertEqual
  ["2","3","5","7","11"]
  (list
    (Cat (Cat (Cat (Single "2") (Single "3")) (Single "5")) (Cat (Single "7") (Single "11")))
  )
test33 :: Assertion
test33 = assertEqual
  [
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11],
    [2,3,5,7,11]
  ]
  (listMap list firstFivePrimeLeafTrees)
```

list is an information losing function, all tree structures in *firstFivePrimeLeafTrees* result in the same *List*. Conversly *sequ* requires information, a strategy, to build *Sequ*s from a *List*.

Of the trees in *firstFivePrimeLeafTrees* we can name two patterns: *Right Branching* and *Left Branching*. A third possible pattern, *Balanced*, is only available for lists with an even number of elements, but we could define a function *leastDepth*. The particular strategy we choose depends upon what use we are to make of the *Sequ* we create, for searching a *leastDepth* strategy is most efficient, for conversion back to a *List* an unbalanced strategy is more efficient.

```
rsequ :: [a] → Sequ a
rsequ [] = Empty
rsequ [x] = Single x
rsequ xs = Cat (rsequ (allButLast xs)) (Single (listLast xs))

allButLast :: [a] → [a]
allButLast [] = []
```

```
allButLast [_] = []
allButLast (x : xs) = x : allButLast xs
test35 :: Assertion
test35 = assertEqual (Cat (Cat (Single 'a') (Single 'b')) (Single 'c'))
  (rsequ "abc")
lsequ :: [a] → Sequ a
lsequ [] = Empty
lsequ [x] = Single x
lsequ (x : xs) = Cat (Single x) (lsequ xs)
test34 :: Assertion
test34 = assertEqual (Cat (Single 'a') (Cat (Single 'b') (Single 'c')))
  (lsequ "abc")
sequ :: [a] → Sequ a
sequ = lsequ
```

Both *list* and *sequ* run in linear time.

1.7 Expression normalisation

```
normalise :: Sequ a → Sequ a
normalise Empty = Empty
normalise (Single x) = Single x
normalise (Cat s Empty) = normalise s
normalise (Cat Empty s) = normalise s
normalise (Cat a b) = Cat (normalise a) (normalise b)
test36 :: Assertion
test36 = assertEqual
    (Cat (Cat (Cat (Cat (Single '2') (Single '3')) (Single '5'))
      (Single '7')) (Single '9'))
  (normalise
    (Cat (Cat (Cat (Cat (Single '2') (Single '3')) (Single '5'))
      (Single '7')) (Single '9')))
test37 :: Assertion
test37 = assertEqual
  (Cat (Cat (Single '1') (Single '2')) (Single '3'))
  (normalise (Cat (Cat (Cat (Single '1') Empty)
      (Cat (Single '2') Empty)) (Cat Empty (Single '3'))))
```

Assuming that *a* is an instance of Eq

```
normalised :: (Eq a) ⇒ Sequ a → Bool
normalised s = s ≡ normalise s
test38 :: Assertion
test38 = assertEqual True
  (normalised (Single 'c'))
```

1.8 Code critique

$$
\begin{aligned}
&badList\ (Empty) &&= [\,]\\
&badList\ (Single\ x) &&= [x]\\
&badList\ (Cat\ (Empty)\ zs) &&= badList\ zs\\
&badList\ (Cat\ (Single\ x)\ zs) &&= x : badList\ zs\\
&badList\ (Cat\ (Cat\ xs\ ys)\ zs) &&= badList\ (Cat\ (Cat\ xs\ ys)\ zs)
\end{aligned}
$$

badList does not have a type declaration, which is considered good style. *badList* is total as the first argument to the constructor. *Cat* in equations 3, 4 and 5 enumerates all possible *Sequ* patterns and equations 1 and 2 match the two possible terminal *Sequ*s.

badList is non-terminating as the left and right sides of equation 5 are the same. For a function to be terminating a recursive call must have a reducing term.

For *Sequ* on which *badList* terminates it is as efficient as possible, that is it runs in linear time for patterns which do not match equation 5.

The longest *Sequ* for which *badList* terminates contains three leaves and is of the form:

```
(Cat (Single 2) (Cat (Single 3) (Single 5)))
```

$$
\begin{aligned}
&nonTerminating :: Assertion\\
&nonTerminating = assertEqual\\
&\quad [2,3,5 :: Int]\\
&\quad (badList\\
&\qquad (Cat\ (Cat\ (Single\ 2)\ (Single\ 3))\ (Single\ 5))\\
&\quad)
\end{aligned}
$$

2 Part II

XHTML is the name given to a number of XML definitions from the W3C. We will use *XHTML 1.0 Strict* [XHTML]. We aim to be able to represent XHTML documents and render them as trees of *Sequ* which we check by converting to text and validating against the W3C validator [W3C Validator].

Haskell has a *Text.Xhtml* module which is descended from *Text.Html* which is in turn derived from [Wallace, Runciman 1999].

2.1 Representing XHTML

XHTML documents are a tree of nested elements. A dubious design decision is to represent text as an element, it would perhaps be better to introduce a super type *XHTMLFragment* which could be *XHTMLText* or *XHTMLText*, as we could then eliminate two *error* calls.

```
  -- Attributes are (name:value) pairs.
data Attribute = Attr String String deriving (Eq)
infixl 4 ===
(===) :: String → String → Attribute
(===) = Attr
  -- An element is text or tagged
data XHTMLElement = XHTMLText String |
  XHTMLElement {tag :: String,
    attributes :: [Attribute],
    content :: Sequ XHTMLElement
    } deriving (Eq)
  -- A page is a sequence of elements
data XHTMLPage = XHTMLPage (Sequ XHTMLElement)
  -- Append XHTMLElement to the current Sequ of content elements
nest :: XHTMLElement → XHTMLElement → XHTMLElement
  -- FIXME This points to the need to introduce a super type
nest (XHTMLText _) _ = error "XHTMLText may not be nested"
nest XHTMLElement {tag = t, attributes = a, content = c} x =
  XHTMLElement {tag = t, attributes = a, content = n} where n = c ++ (Single x)
infixl 2 <<<
(<<<) :: XHTMLElement → XHTMLElement → XHTMLElement
(<<<) = nest
  -- Add attribute (prepend to attribute list)
addAttribute :: XHTMLElement → Attribute → XHTMLElement
addAttribute XHTMLElement {tag = t, attributes = as, content = c} a =
  XHTMLElement {tag = t, attributes = a : as, content = c}
    -- FIXME This points to the need to introduce a super type
addAttribute (XHTMLText _) _ = error "XHTMLText may not have attributes added"
infixl 3 @@@   -- bind more closely than nesting
(@@@) :: XHTMLElement → Attribute → XHTMLElement
```

```
(@@@) = addAttribute
```

2.2 Outputting XHTML from the representation

An XML document is composed of an XML declaration, which determines the character set to be used, an XML DOCTYPE tag which defines the Document Type Definition (DTD) and XML elements.

```
  -- Show name unquoted, value quoted, with spaces before and after
instance Show Attribute where
  show (Attr k v) = "  " +++ k +++ "=" +++ show v +++ "  "
  showList [] = showString ""
  showList attrs = showString (listConcatMap show attrs)
```

An XML element typically consists of a start tag, content and an end tag. A start tag consists of a label and attributes. Element content may be empty or contain a sequence of text or elements. An end tag consists of the tag label eg `</tag>`. There is a special syntax for an empty element, where an end tag is not required eg `<tag/>`.

```
  -- Show an element as text or a possibly empty element
instance Show XHTMLElement where
  show (XHTMLText s) = s
  show (XHTMLElement {tag = t, attributes = attrs, content = Empty}) =
    "<" +++ t +++ show attrs +++ "/>"
  show (XHTMLElement {tag = t, attributes = attrs, content = c}) =
    "<" +++ t +++ show attrs +++ ">" +++ (consume show (+++) "" c) +++
    "</" +++ t +++ ">\n"
  -- Prepend XML and DTD declarations then show tree
instance Show XHTMLPage where
  show (XHTMLPage xs) = xmlDefinition
    +++ "\n"
    +++ documentTypeDefinition
    +++ "\n"
    +++ (consume show (+++) "\n" xs)
    +++ "\n"
  -- required at top of document
xmlDefinition :: String
xmlDefinition = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
  -- required immediately after XML definition
documentTypeDefinition :: String
documentTypeDefinition = "<!DOCTYPE html PUBLIC "
  +++ ['"'] +++ "-//W3C//DTD XHTML 1.0 Strict//EN"
  +++ ['"',' ','"']
  +++ "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
  +++ ['"',' ','>']
  -- Add enclosing html element
xhtmlPage :: XHTMLPage -> XHTMLPage
xhtmlPage (XHTMLPage els) = XHTMLPage (Single XHTMLElement {tag = "html",
```

```
    attributes = ["xmlns" === "http://www.w3.org/1999/xhtml"],
    content = els})

  -- Create an html page from a head and body
showXHTMLPage :: XHTMLPage → Sequ Char
showXHTMLPage xPage = sequ (show $ xhtmlPage xPage)
```

Text may be characters from our specified character set (UTF-8) or entites representing characters; where entities are a string of lowercase letters prefaced with an `&` and terminated with a `;` eg `&`.

```
  -- if the text contains markup characters escape them to entities
xhtmlEscape :: String → XHTMLElement
xhtmlEscape s = XHTMLText (listConcatMap tr s)

tr :: Char → String
tr '<' = "&lt;";
tr '>' = "&gt;"
tr '&' = "&amp;"
tr c = [c]

  -- Used when the text contains entities ie is already escaped
escaped :: String → XHTMLElement
escaped s = XHTMLText s
test39 :: Assertion
test39 = assertEqual (escaped "&lt;p&gt;B&amp;O&lt;/p&gt;")
  (xhtmlEscape "<p>B&O</p>")
test40 :: Assertion
test40 = assertEqual
  (Cat (Single 'D') (Cat (Single 'o') (Single 'g')))
  (sequ "Dog")
charSequToString :: Sequ Char → String
charSequToString Empty = []
charSequToString (Single c) = [c]
charSequToString (Cat l r) = (charSequToString l) +++
  (charSequToString r)
test41 :: Assertion
test41 = assertEqual
  "abc"
  (charSequToString (Cat (Cat (Single 'a') (Single 'b')) (Single 'c')))
```

2.2.1 Element creation functions

```
  -- Create a simple text element
element :: String → String → XHTMLElement
element tName s = XHTMLElement
  {tag = tName,
                  attributes = [],
                  content = (Single $ xhtmlEscape s)}

  -- Create a title
```

```
title :: String → XHTMLElement
title s = element "title" s

  -- Create an h1
h1 :: String → XHTMLElement
h1 s = element "h1" s

  -- Create a p
p :: String → XHTMLElement
p s = element "p" s

  -- Create an img
img :: String → String → String → String →
  XHTMLElement
img src alt height width = XHTMLElement {
  tag = "img",
  attributes = ["src" === src,
    "alt"          === alt,
    "title"        === alt,   -- repeated
    "height"       === height,
    "width"        === width
    ],
  content = Empty}
  -- Create an anchor with required attribute href
anchor :: String → XHTMLElement → XHTMLElement
anchor url e = XHTMLElement {
  tag = "a",
  attributes = ["href" === url],
  content = (Single e)}
```

2.2.2 Creating a specific XHMTL instance

```
testText :: String
testText = "Hello World <& your dog>"
xhtmlTestText :: XHTMLElement
xhtmlTestText = xhtmlEscape testText
lorem :: XHTMLElement
lorem = p (listConcat ["Lorem ipsum dolor sit amet, consectetur adipiscing ",
  "elit. Sed viverra tellus lacus. Curabitur tempus auctor",
  "est, pellentesque posuere turpis rhoncus eget. ",
  "Pellentesque lacus mi, consectetur et posuere eget, ",
  "porta a sem. Aenean vel dictum enim. Pellentesque et ",
  "velit ipsum, eu tempor felis. Suspendisse felis tellus, ",
  "posuere sit amet ultricies sit amet, luctus id neque. ",
  "Donec nulla turpis, tempor a tincidunt nec, eleifend ",
  "et turpis. In eget turpis eu nisl consequat gravida sit ",
  "amet sit amet urna."])
helloImg :: XHTMLElement
helloImg = (img "hello.jpg" (show xhtmlTestText) "78" "298")
```

```
    @@@ "onmouseover" === "style.border='2px solid red';"
    @@@ "onmouseout" === "style.border='2px solid green'"
    @@@ "style"      === "border:2px solid green;"
imgP :: XHTMLElement
imgP = XHTMLElement {
  tag = "p",
  attributes = [],
  content = (Single (anchor "http://google.com" helloImg))}
br :: XHTMLElement
br = XHTMLElement {
  tag = "br",
  attributes = [],
  content = Empty}
hr :: XHTMLElement
hr = XHTMLElement {
  tag = "hr",
  attributes = ["style" === "border:2px solid blue;"],
  content = Empty}
validP :: XHTMLElement
validP = XHTMLElement {
  tag = "p",
  attributes = [],
  content = (Cat (Single br)
    (Single (anchor "http://validator.w3.org/check?uri=referer"
      (img "http://www.w3.org/Icons/valid-xhtml10"
        "Valid XHTML 1.0 Strict" "31" "88"))))}
divBlock :: XHTMLElement
divBlock = XHTMLElement {
  tag = "div",
  attributes = ["style" === "text-align:left; width:640px;"],
  content = Empty} <<< h1 testText <<< imgP <<< lorem <<< hr
bodyBlock :: XHTMLElement
bodyBlock = XHTMLElement {
  tag = "body",
  attributes = ["onload" === ("alert('" +++ (show xhtmlTestText) +++ "');"),
    "style" === "text-align:center;width:100%"],
  content = (Single divBlock)} <<< validP
headBlock :: XHTMLElement
headBlock = XHTMLElement {
  tag = "head",
  attributes = [],
  content = (Single $ title testText)}
testPage :: XHTMLPage
testPage = XHTMLPage (Cat (Single headBlock) (Single bodyBlock))
test42 :: Assertion
```

```
test42 = assertEqual
  (listConcat ["<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n",
    "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" ",
    "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\" >\n",
    "<html xmlns=\"http://www.w3.org/1999/xhtml\" >",
    "<head><title>Hello World &lt;&amp; your dog&gt;</title>\n",
    "</head>\n",
    "<body onload=\"alert('Hello World &lt;&amp; your dog&gt;');\"",
    "  style=\"text-align:center;width:100%\" >",
    "<div style=\"text-align:left; width:640px;\" >",
    "<h1>Hello World &lt;&amp; your dog&gt;</h1>\n",
    "<p><a href=\"http://google.com\" >",
    "<img ",
    "style=\"border:2px solid green;\"  ",
    "onmouseout=\"style.border='2px solid green'\"  ",
    "onmouseover=\"style.border='2px solid red';\"  ",
    "src=\"hello.jpg\"  ",
    "alt=\"Hello World &lt;&amp; your dog&gt;\"  ",
    "title=\"Hello World &lt;&amp; your dog&gt;\"  ",
    "height=\"78\"  width=\"298\" ",
    "/></a>\n",
    "</p>\n",
    "<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. ",
    "Sed viverra tellus lacus. Curabitur tempus auctorest, ",
    "pellentesque posuere turpis rhoncus eget. ",
    "Pellentesque lacus mi, consectetur et posuere eget, ",
    "porta a sem. Aenean vel dictum enim. Pellentesque et velit ",
    "ipsum, eu tempor felis. Suspendisse felis tellus, posuere ",
    "sit amet ultricies sit amet, luctus id neque. ",
    "Donec nulla turpis, tempor a tincidunt nec, eleifend et ",
    "turpis. In eget turpis eu nisl consequat gravida sit amet ",
    "sit amet urna.</p>\n",
    "<hr style=\"border:2px solid blue;\" /></div>\n",
    "<p><br/>",
    "<a href=\"http://validator.w3.org/check?uri=referer\" >",
    "<img src=\"http://www.w3.org/Icons/valid-xhtml10\"  ",
    "alt=\"Valid XHTML 1.0 Strict\"  ",
    "title=\"Valid XHTML 1.0 Strict\"  ",
    "height=\"31\"  width=\"88\" /></a>\n",
    "</p>\n",
    "</body>\n",
    "</html>\n\n"
    ])
    (charSequToString (showXHTMLPage testPage))
```

2.2.3 Outputting an XHMTL file

If all tests pass then the page is written to the file system.

```
output :: IO ()
output = do
  writeFile "fpr.xhtml" (charSequToString (showXHTMLPage testPage))
```

The resulting xhtml file can be seen at [output] and is shown valid at [W3C Validator].

2.3 Conclusion

The model and its dependant output functions can represent valid XHTML documents. It can also represent invalid documents (by for example placing an *XHTMLText* element directly inside a *body* element). The representaton does nothing to check that Javascript or CSS is valid.

One could extend this project to parse a DTD to generate element creation functions with the appropriate required attributes; the augmentation function (*addAttribute*) can be used to handle optional attributes. This would also enable validation of attribute names and simple type checking on attribute values.

One could also envisage extending the project to parse XHTML into the representation so enabling round-tripping.

3 Part III

3.1 Functional Programming and Java™

The history of programing languages is convoluted, with functional, imperative and object based languages being intertwined, at least in time, and with the 'best bits', once they have been originated, being used by all subsequent languages (by definition). Both functional languages and non-functional languages owe an enormous debt to John McCarthy who both invented Lisp and served on the Algol committee from which most subsequent imperative languages derived[Graham 2001].

A language is not a single thing, so *Java 1.1* is quite different to *Java 8*, however the requirement for backward compatibility has led Java to evolve into a very complex language.

There is a set of features, such as *Garbage Collection* [Stutter 2011], which were first introduced in functional languages and are used in main-stream languages such as Java™, but will not be considered to be *Functional Programming Features* but merely common building blocks first demonstrated within a functional framework. Similarly we will not count *Conditionals* or *variables as pointers* which were all first demonstrated in *Lisp* [Graham 2001].

3.1.1 Function Types

The defining feature of a functional programming language is function types, or *having functions as first class objects*. *Java™* does not have *Function Types*, relying upon anonymous classes and introspection.

3.1.2 Lambdas

Java version 1 - 7 do not have a built in syntax for lambdas. Anonymous Inner Classes are closures which *Java™* has always had. [java.lang.Thread] and more recently [java.util.concurrent.Executor] can take a closure as an argument and execute it

```
public class Hello {
  public static void main(String[] args) {
    new Thread(
        new Runnable() {
          public void run() {
            System.out.println("Hello World");
          }
        }).start();
  }
}
```

`Runnable` is a Interface with a Single Abstract Method (SAM), and it is expected that lambdas will be released in Java8 (currently scheduled for Summer 2013 [Java8 Release]) based upon SAM interfaces [State of the Lambda]. Currently variables defined outside the scope of an anonymous inner class must be declared final for them to be accessed inside. The lambdas in Java8 will not have this requirement, however any attempt

to access a non-final variable declared outside the scope of a lambda will result in an exception [Langer]. A class will only be allowed to contain one lambda, should you require more then you must revert to anonymous inner classes [Langer]. Given that *C#* now has a well worked out syntax for lambdas the introduction of lambda to *Java*^{TM} is too little too late.

As an aside, *C++* has had lambdas of the following form

```
#include <iostream>

int main(){
  auto lambda=[]{ std::cout << "Hello Lambda" << std::endl;};
  lambda();
}
```

since the *C++ 11* standard was approved by ISO on 12 August 2011 [Sommerlad]. So we are in the surprising position that *C++* has lambdas before *Java*^{TM}; justifying the Haskell slogan "avoid sucess at all costs".

3.1.3 Static Typing

Java and Haskell both have strong static typing, that is the compiler can detect type errors. Haskell does not have a base object type and so objects cannot be coerced to other types, where Java can, leading to the possibility of type cast exceptions.

3.1.4 Polymorphism and Generics

Generics were introduced into Java5 to enable polymorphic methods. Generic types are however removed by the compiler (type erasure). Implementations of generic interfaces still require a non-generic method to be generated by the compiler (bridging methods). These methods are not available to the programmer, other than by introspection, leading test coverage tools to marks classes as less than 100% covered. "Generics are a nasty business"[Langer].

3.1.5 Type inference

The generics sytem does have a form of type inference, but the original syntax is still in place leading to the verbose, redundant syntax of

```
Object o = new Object();
```

In Java7 some type inference has been introduced with the diamond operator [Goetze] which reduces the redundancy:

```
public List<String> list = new ArrayList<>();
```

3.1.6 Auto-boxing and primitive types

Both Haskell and *Java*^{TM} make a distinction between primitive data types such as *int*, *boolean*, *double* and objects [Haskell Primitives] and place the same restictions on them. In particular a container (or box) may not contain primitives.

This distinction is made for pragmatic reasons, objects being considered unnecessarily heavyweight and a three fold improvement is still available in Haskell for numerically intensive programs [Haskell Primitives].

The two types of data type in Java is now considered confusing and result in cluttered boilerplate code when, for example, storing primitive data types in a *Collection* so auto-boxing was introduced in Java5 [Auto-Boxing]. This is only a sticking plaster and can lead to confusion as to what type is actually being dealt with.

3.1.7 Handling bottom ($\perp$)

The *undefined* state can be reached in a number of ways, for example infinite loops, non-reducing recursion, division by zero,

In *Java*^{TM} every object can be *null* this is equvalent to "adding an implicit `Maybe a` to every type `a`" [StackOverflow 1], which means that every dereference has to check for null [SpecialCase]. The introduction of *null* has been called by Tony Hoare, who introduced *null* references into ALGOL in 1965, his "billion dollar mistake" [Hoare 2010].

3.1.8 Exception handling

Java `Exception` (checked exceptions) and Haskell's `Control.Exception` are equivalent. It is argued that checked exceptions have been overused Java, especially in relation to `IOException`, where the programmer can often do nothing to recover the situation. This is not really the motivation for preferring `RuntimeException` (unchecked exceptions). It is that Java forces checked exceptions into the method signature, so forcing the programmer to write boilerplate code. This is dangerous, as Java programmers increasingly use `RuntimeException` when they should in fact be using an `Exception`. From a functional programming perspective it is always preferable for the type to reflect the model and to allow the programmer to handle failure than to return `error` ($\perp$) from an arbitrary point in the code.

3.1.9 Recursion

Java^{TM} has both recursive function calls and recursive data type definitions.

3.1.10 Lazy Evaluation

Whilst *Lazy Evaluation* can be achieved in Java^{TM} it is painfully verbose (approximately ten times "larger" than in a functional language [Dekker 2006]).

3.1.11 List Comprehensions

List comprehensions, originated in the functional language NPL [Darlington 1977] have since been used in other languages such as Miranda, Haskell, Erlang, Python [Peyton Jones, Wadler 2007
A list comprehension can be written in Java as an enumeration but there is no syntactic assistance.

```
public class FromToEnumeration
    implements java.util.Enumeration<Integer> {
```

```
  private Integer current;
  private Integer to;

  public FromToEnumeration(Integer from, Integer to) {
    this.current = from - 1;
    this.to = to;
  }

  public boolean hasMoreElements() {
    return (current < to) ? true : false;
  }

  public Integer nextElement() {
    return ++current;
  }
}
```

3.1.12 Purity

Haskell is a *pure* functional language. Purity here is used in the same sense as the *PURE* keyword in Fortran [Fortran 2010 Draft]. This means that functions may not have side effects and their value must depend soley upon their arguments, and hence may be executed in any order. Unfortunately Java™ does not have a mechanism for ensuring function purity, though add-ons and subsets have been proposed [Joe-E].

3.1.13 Conclusion

Language design must take into account human use, mathematical properties and machine execution.

A language must be learnable but the ability to second guess what makes a language easy or difficult for the novice has proved elusive, as so definitively shown by BASIC. I would argue that the historical pedagogical order of imperative, object oriented and finally functional programing languages is exactly wrong.

Another human factor to be taken into account is ease of debugging. Functional programming features such as stong static typing and purity remove whole classes of elusive bugs. However a division within the functional programming languages, between Haskell and ML for example, is the distance from the underlying execution model of the machine running the program.

The mathematical properties of a language are important if one wishes to prove programs correct or to generate correct programs.

The current emphasis within computing for interconnected processors, either over a network or within a multi-cpu machine, requires that functions be *pure*.

For me functional programming has been the missing piece of the jigsaw and I am very grateful for the opportunity to have been taught it.

References

[Auto-Boxing] Autoboxing
`http://download.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html`

[Binary Trees] Catalan Binary Trees
`http://hackage.haskell.org/packages/archive/combinat/0.2.4/doc/html/Math-Combinat-Trees-Binary.html`

[Bird 1998] Richard Bird
Introduction to Functional Programming using Haskell
ISBN 0-13-484346-0

[Classes] Classes
`http://www.haskell.org/tutorial/classes.html`

[Darlington 1977] John Darlington,
Program Transformation and Synthesis: Present Capabilities
Research Report No. 77/43, Dept. of Computing and Control, Imperial College of Science and Technology, London September 1977.

[Dekker 2006] Anthony H. Dekker
Lazy Functional Programming in Java ACM SigPLAN notices
Vol. 41 (3) March 2006

[Fortran 2010 Draft] Information technology *Programming languages* Fortran
ISO/IEC DIS 1539-1 Draft
Section 12.7
`ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf`

[FPR 2011] Course Material
Functional Programming, FPR
Software Engineering Programme,
Oxford University,
October 2011

[State of the Lambda] Brian Goetze
State of the Lambda
`http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-3.html`

[Goetze] Brian Goetze
Language designer's notebook: Quantitative language design
`http://www.ibm.com/developerworks/java/library/j-ldn1/`

[Graham 2001] Paul Graham
What Made Lisp Different
2001 `http://www.paulgraham.com/diff.html`

[Haskell Primitives] *Haskell Primitives*
`http://www.haskell.org/ghc/docs/6.8.1/html/users_guide/primitives.html`

[Hinze,Ross 2006] Ralf Hinze and Ross Paterson
Journal of Functional Programming16:2 (2006), pages 197-217
`http://www.soi.city.ac.uk/~ross/papers/FingerTree.html`.

[Hoare 2010] Tony Hoare
Null References: The Billion Dollar Mistake
QCON 2009
`http://qconlondon.com/london-2009/presentation/Null+References:+The+Billion+Dollar+Mistake`

[Hughes 1984] John Hughes
Why Functional Programming Matters
Institutionen för Datavetenskap,
Chalmers Tekniska Högskola,
41296 Göteborg,
SWEDEN.
`http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf`

[HUnit] HUnit
`http://hunit.sourceforge.net/`

[java.lang.Thread] java.lang.Thread
`http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html`

[java.util.concurrent.Executor] java.util.concurrent.Executor
`http://download.oracle.com/javase/1,5.0/docs/api/java/util/concurrent/Executor.html`

[Java8 Release] *Java One keynote slide*
`http://blogs.oracle.com/javaone/resource/java_keynote/slide_16_full_size.gif`

[Joe-E] Matthew Finifter, Adrian Mettler, Naveen Sastry, David Wagner
Department of Electrical Engineering and Computer Science
University of California, Berkeley, USA
Veriable Functional Purity in Java
`http://www.cs.berkeley.edu/~amettler/pure-ccs08.pdf`

[Langer] Angelika Langer
What's new in Java7
ACCU 2011 Conference, Oxford

[output] *Output from this file*
`http://context-computing.co.uk/FPR/fpr.xhtml`

[Peyton Jones, Wadler 2007] Simon Peyton Jones, Philip Wadler
Comprehensive Comprehensions
Haskell'07, Freiberg Germany September 2007

[Sommerlad] Prof. Peter Sommerlad
Response on accu-general maiing list - 2011-11-19
Institut fr Software: Bessere Software - Einfach, Schneller!
HSR Hochschule fr Technik Rapperswil
Oberseestr 10, Postfach 1475, CH-8640 Rapperswil

[Snape 2005] Jamie Snape
Loopless Functional Algorithms
MSc thesis September 2005, University of Oxford
`http://wwwx.cs.unc.edu/~snape/publications/msc/thesis.pdf`

[SpecialCase] Martin Fowler
Special Case
`http://martinfowler.com/eaaCatalog/specialCase.html`

[StackOverflow 1] *What's the difference between undefined in Haskell and null in Java?*
`http://stackoverflow.com/questions/3962939/whats-the-difference-between-undefined-in-haskell-and-null-in-java`

[Stutter 2011] Herb Stutter
John McCarthy
`http://herbsutter.com/2011/10/25/john-mccarthy/`

[Tate 2010] Bruce A. Tate
Seven Languages in Seven Weeks
ISBN-13: 978-1-934356-59-3

[Wallace, Runciman 1999] Malcolm Wallace, Colin Runciman
Haskell and XML: Generic Document Processing Combinators vs. Type-Based Translation
`http://xml.coverpages.org/wallace-haskell-paper.html`

[W3C Validator] *Output of W3C Validator Service* `http://validator.w3.org/check?uri=http%3a%2f%2fcontext-computing%2eco%2euk%2fFPR%2ffpr%2exhtml`

[XHTML] *XHTML-1.0-Strict* `http://www.w3.org/TR/xhtml1/dtds.html#a_dtd_XHTML-1.0-Strict`