1 Prelude

1.1 Standard types and classes

1.1.1 Folds and traversals

```
-- [Int] -> Int
sum = foldl (+) 0

-- [Int] -> Int
product = foldl (*) 1

-- [Int] -> Int
maximum [x] = x
maximum (x:xs) = max x (maximum xs)

-- [Int] -> Int
minimum [x] = x
minimum (x:xs) = min x (minimum xs)
```

1.2 List operations

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

-- [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)

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

-- (a -> Bool) -> [a] -> [a]
filter p []   = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs

-- [a] -> a
head (x:_) = x

-- [a] -> [a]
tail (_:xs) = xs

-- [a] -> a
last [x]    = x
last (_:xs) = last xs

-- [a] -> [a]
init [x]    = []
init (x:xs) = x : init xs

-- t a -> Bool
null = foldr (\_ _ -> False) True

-- t a -> Int
length = foldl' (\c _ -> c+1) 0

-- [a] -> [a]
reverse = foldl (flip (:)) []
```

1.2.1 Special folds

```
-- [Bool] -> Bool
and = foldr (&&) True

-- [Bool] -> Bool
or = foldr (||) False

-- (a -> Bool) -> [a] -> Bool
any p xs = or (map p xs)

-- (a -> Bool) -> [a] -> Bool
any p xs = or (map p xs)

-- [[a]] -> [a]
concat = foldr (++) []

-- (a -> [b]) -> [a] -> [b]
concatMap f = foldr ((++) . f) []
```

1.2.2 Building lists with scans

```
-- (b -> a -> b) -> b -> [a] -> [b]
scanl = s'
  where
    -- (b -> a -> b) -> b -> [a] -> [b]
    s' f q ls = q : (case ls of
      [] -> []
      x:xs -> s' f (f q x) xs)

-- (a -> a -> a) -> [a] -> [a]
scanl1 _ []     = []
scanl1 f (x:xs) = scanl f x xs

-- (a -> b -> b) -> b -> [a] -> [b]
scanr _ q0 []     = [q0]
scanr f q0 (x:xs) = f x q : qs
  where qs@(q:_) = scanr f q0 xs

-- (a -> a -> a) -> [a] -> [a]
scanr1 _ []     = []
scanr1 _ [x]    = [x]
scanr1 f (x:xs) = f x q : qs
  where qs@(q:_) = scanr1 f xs
```

1.2.3 Building infinite lists

```
-- (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

-- a -> [a]
repeat x = xs where xs = x : xs

-- Int -> a -> [a]
replicate n x = take n (repeat x)

-- [a] -> [a]
cycle xs = xs' where xs' = xs ++ xs'
```

1.2.4 Sublists

```
-- Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []         = []
take n (x:xs)     = x : take (n-1) xs

-- Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []          = []
drop n (_:xs)      = drop (n-1) xs

-- Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)

-- (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
  | p x       = x : takeWhile p xs
  | otherwise = []

-- (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p xs@(x:xs')
  | p x       = dropWhile p xs'
  | otherwise = xs

-- (a -> Bool) -> [a] -> ([a], [a])
span _ []     = ([], [])
span p xs@(x:xs')
  | p x       = (x:ys, zs)
  | otherwise = ([], xs)
    where (ys, zs) = span p xs'

-- (a -> Bool) -> [a] -> ([a], [a])
break p = span (not . p)
```

1.2.5 Searching lists

```
-- (Foldable t, Eq a) => a -> t a -> Bool
notElem x = not . elem x

-- (Eq a) => a -> [(a, b)] -> Maybe b
lookup key []    = Nothing
lookup key ((x, y):xys)
    | key == x  = Just y
    | otherwise = lookup key xys
```

1.2.6 Zipping and unzipping lists

```
-- [a] -> [b] -> [(a, b)]
zip []     bs     = []
zip as     []     = []
zip (a:as) (b:bs) = (a, b) : zip as bs

-- (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f []     bs     = []
zipWith f as     []     = []
zipWith f (a:as) (b:bs) =
  f a b : zipWith f as bs

-- [(a, b)] -> ([a], [b])
unzip = foldr f ([], [])
  where
    f (a, b) ~(as, bs) = (a:as, b:bs)
```

See also: zip3, zipWith3 and unzip3.

1.2.7 Functions on strings

```
-- String -> [String]
lines "" = []
lines s  = cons (case break (== '\n') s of
  (l, s') -> (l, case s' of
    []    -> []
    _:s'' -> lines s''))
  where
    cons ~(h, t) = h : t

-- [String] -> String
unlines = concatMap (++ "\n")

-- String -> [String]
words s = case dropWhile Char.isSpace s of
  "" -> []
  s' -> w : words s''
    where (w, s'') = break Char.isSpace s'

-- [String] -> String
unwords [] = ""
unwords ws = foldr1 (\w s -> w ++ ' ':s) ws
```

2 Data.List

2.1 Special lists

2.1.1 Set operations

```
-- (Eq a) => [a] -> [a]
nub = nubBy (==)

nubBy p []     = []
nubBy p (x:xs) = x :
  nubBy p (filter (\ y -> not (p x y)) xs)
```

2.1.2 Ordered lists

```
-- (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) =
  let large = sort $ filter (> x) xs
      small = sort $ filter (<= x) xs
  in small ++ ([x]:large)
```

In this document almost entire Haskell Prelude is given. Many of the definitions are written with clarity rather than efficiency in mind!

author: Remigiusz Suwalski,
date: September 13, 2017