JFP **13** (1): 179–190, January 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803001916 Printed in the United Kingdom

Chapter 17

List Utilities

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\), deleteFirstsBy,
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,
    -- ...and what the Prelude exports
    -- [](:), []),     -- This is built-in syntax
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

infix 5 \\
```

```
elemIndex           :: Eq a => a -> [a] -> Maybe Int
elemIndices         :: Eq a => a -> [a] -> [Int]
find                :: (a -> Bool) -> [a] -> Maybe a
findIndex           :: (a -> Bool) -> [a] -> Maybe Int
findIndices         :: (a -> Bool) -> [a] -> [Int]
nub                 :: Eq a => [a] -> [a]
nubBy               :: (a -> a -> Bool) -> [a] -> [a]
delete              :: Eq a => a -> [a] -> [a]
deleteBy            :: (a -> a -> Bool) -> a -> [a] -> [a]
(\\)                :: Eq a => [a] -> [a] -> [a]
deleteFirstsBy      :: (a -> a -> Bool) -> [a] -> [a] -> [a]
union               :: Eq a => [a] -> [a] -> [a]
unionBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]

intersect           :: Eq a => [a] -> [a] -> [a]
intersectBy         :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersperse         :: a -> [a] -> [a]
transpose           :: [[a]] -> [[a]]
partition           :: (a -> Bool) -> [a] -> ([a],[a])
group               :: Eq a => [a] -> [[a]]
groupBy             :: (a -> a -> Bool) -> [a] -> [[a]]
inits               :: [a] -> [[a]]
tails               :: [a] -> [[a]]
isPrefixOf          :: Eq a => [a] -> [a] -> Bool
isSuffixOf          :: Eq a => [a] -> [a] -> Bool
mapAccumL           :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR           :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
unfoldr             :: (b -> Maybe (a,b)) -> b -> [a]
sort                :: Ord a => [a] -> [a]
sortBy              :: (a -> a -> Ordering) -> [a] -> [a]
insert              :: Ord a => a -> [a] -> [a]
insertBy            :: (a -> a -> Ordering) -> a -> [a] -> [a]
maximumBy           :: (a -> a -> Ordering) -> [a] -> a
minimumBy           :: (a -> a -> Ordering) -> [a] -> a
genericLength       :: Integral a => [b] -> a
genericTake         :: Integral a => a -> [b] -> [b]
genericDrop         :: Integral a => a -> [b] -> [b]
genericSplitAt      :: Integral a => a -> [b] -> ([b],[b])
genericIndex        :: Integral a => [b] -> a -> b
genericReplicate    :: Integral a => a -> b -> [b]

zip4                :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip5                :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip6                :: [a] -> [b] -> [c] -> [d] -> [e] -> [f]
                          -> [(a,b,c,d,e,f)]
zip7                :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
                          -> [(a,b,c,d,e,f,g)]
zipWith4            :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith5            :: (a->b->c->d->e->f) ->
                       [a]->[b]->[c]->[d]->[e]->[f]
zipWith6            :: (a->b->c->d->e->f->g) ->
                       [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7            :: (a->b->c->d->e->f->g->h) ->
                       [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip4              :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip5              :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip6              :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip7              :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

This library defines some lesser-used operations over lists.

17.1 Indexing Lists

- `elemIndex val list` returns the index of the first occurrence, if any, of `val` in `list` as `Just index`. `Nothing` is returned if `not (val `elem` list)`.
- `elemIndices val list` returns an in-order list of indices, giving the occurrences of `val` in `list`.
- `find` returns the first element of a list that satisfies a predicate, or Nothing, if there is no such element. `findIndex` returns the corresponding index. `findIndices` returns a list of all such indices.

17.2 "Set" Operations

There are a number of "set" operations defined over the `List` type. `nub` (meaning "essence") removes duplicates elements from a list. `delete`, `(\\)`, `union` and `intersect` (and their `By` variants) preserve the invariant that their result does not contain duplicates, provided that their first argument contains no duplicates.

- `nub` removes duplicate elements from a list. For example:

```
nub [1,3,1,4,3,3] = [1,3,4]
```

- `delete x` removes the first occurrence of `x` from its list argument, e.g.

```
delete 'a' "banana" == "bnana"
```

- `(\\)` is list difference (non-associative). In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus, `(xs ++ ys) \\ xs == ys`.
- `union` is list union, e.g.

```
"dog" `union` "cow" == "dogcw"
```

- `intersect` is list intersection, e.g.

```
[1,2,3,4] `intersect` [2,4,6,8] == [2,4]
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

17.3 List Transformations

- `intersperse sep` inserts `sep` between the elements of its list argument, e.g.

```
intersperse ',' "abcde" == "a,b,c,d,e"
```

- `transpose` transposes the rows and columns of its argument, e.g.

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

- `partition` takes a predicate and a list and returns a pair of lists: those elements of the argument list that do and do not satisfy the predicate, respectively; i.e.

```
partition p xs == (filter p xs, filter (not . p) xs)
```

- `sort` implement a stable sorting algorithm, here specified in terms of the `insertBy` function, which inserts objects into a list according to the specified ordering relation.
- `insert` inserts a new element into an *ordered* list (arranged in increasing order).
- `group` splits its list argument into a list of lists of equal, adjacent elements. For example

```
group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
```

- `inits` returns the list of initial segments of its argument list, shortest first.

```
inits "abc" == ["","a","ab","abc"]
```

- `tails` returns the list of all final segments of its argument list, longest first.

```
tails "abc" == ["abc", "bc", "c",""]
```

- `mapAccumL f s l` applies `f` to an accumulating "state" parameter `s` and to each element of `l` in turn.
- `mapAccumR` is similar to `mapAccumL` except that the list is processed from right-to-left rather than left-to-right.

17.4 `unfoldr`

The `unfoldr` function is a "dual" to `foldr`: while `foldr` reduces a list to a summary value, `unfoldr` builds a list from a seed value. For example:

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
iterate f == unfoldr (\x -> Just (x, f x))
```

In some cases, `unfoldr` can undo a `foldr` operation:

```
unfoldr f' (foldr f z xs) == xs
```

if the following holds:

```
f' (f x y) = Just (x,y)
f' z       = Nothing
```

17.5 Predicates

`isPrefixOf` and `isSuffixOf` check whether the first argument is a prefix (resp. suffix) of the second argument.

17.6 The "`By`" Operations

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with "`By`". For example, the function `nub` could be defined as follows:

```
nub              :: (Eq a) => [a] -> [a]
nub []           =  []
nub (x:xs)       =  x : nub (filter (\y -> not (x == y)) xs)
```

However, the equality method may not be appropriate in all situations. The function:

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

allows the programmer to supply their own equality test. When the "`By`" function replaces an `Eq` context by a binary predicate, the predicate is assumed to define an equivalence; when the "`By`" function replaces an `Ord` context by a binary predicate, the predicate is assumed to define a total ordering.

The "`By`" variants are as follows: `nubBy`, `deleteBy`, `deleteFirstsBy` (the `By` variant of `\\`), `unionBy`, `intersectBy`, `groupBy`, `sortBy`, `insertBy`, `maximumBy`, `minimumBy`.

The library does not provide `elemBy`, because `any (eq x)` does the same job as `elemBy eq x` would. A handful of overloaded functions (`elemIndex`, `elemIndices`, `isPrefixOf`, `isSuffixOf`) were not considered important enough to have "`By`" variants.

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

17.7 The "generic" Operations

The prefix "generic" indicates an overloaded function that is a generalised version of a Prelude function. For example,

```
genericLength       :: Integral a => [b] -> a
```

is a generalised version of length.

The "generic" operations are as follows: genericLength, genericTake, genericDrop, genericSplitAt, genericIndex (the generic version of !!), genericReplicate.

17.8 Further "zip" Operations

The Prelude provides zip, zip3, unzip, unzip3, zipWith, and zipWith3. The List library provides these same three operations for 4, 5, 6, and 7 arguments.

17.9 Library List

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\), deleteFirstsBy,
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,
    -- ...and what the Prelude exports
    -- [](:), [],     -- This is built-in syntax
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

import Maybe( listToMaybe )
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
infix 5 \\

elemIndex               :: Eq a => a -> [a] -> Maybe Int
elemIndex x             =  findIndex (x ==)

elemIndices             :: Eq a => a -> [a] -> [Int]
elemIndices x           =  findIndices (x ==)

find                    :: (a -> Bool) -> [a] -> Maybe a
find p                  =  listToMaybe . filter p

findIndex               :: (a -> Bool) -> [a] -> Maybe Int
findIndex p             =  listToMaybe . findIndices p

findIndices             :: (a -> Bool) -> [a] -> [Int]
findIndices p xs        =  [ i | (x,i) <- zip xs [0..], p x ]

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

nubBy                   :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []             =  []
nubBy eq (x:xs)         =  x : nubBy eq (filter (\y -> not (eq x y)) xs)

delete                  :: Eq a => a -> [a] -> [a]
delete                  =  deleteBy (==)

deleteBy                :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy eq x []        = []
deleteBy eq x (y:ys)    = if x `eq` y then ys else y : deleteBy eq x ys

(\\)                    :: Eq a => [a] -> [a] -> [a]
(\\)                    =  foldl (flip delete)

deleteFirstsBy          :: (a -> a -> Bool) -> [a] -> [a] -> [a]
deleteFirstsBy eq       =  foldl (flip (deleteBy eq))

union                   :: Eq a => [a] -> [a] -> [a]
union                   =  unionBy (==)

unionBy                 :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys        =  xs ++ deleteFirstsBy eq (nubBy eq ys) xs

intersect               :: Eq a => [a] -> [a] -> [a]
intersect               =  intersectBy (==)

intersectBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys    =  [x | x <- xs, any (eq x) ys]

intersperse             :: a -> [a] -> [a]
intersperse sep []      =  []
intersperse sep [x]     =  [x]
intersperse sep (x:xs)  =  x : sep : intersperse sep xs
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
-- transpose is lazy in both rows and columns,
--       and works for non-rectangular 'matrices'
-- For example, transpose [[1,2],[3,4,5],[]]  =  [[1,3],[2,4],[5]]
-- Note that [h | (h:t) <- xss] is not the same as (map head xss)
--      because the former discards empty sublists inside xss
transpose               :: [[a]] -> [[a]]
transpose []            = []
transpose ([]     : xss) = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:t) <- xss]) :
                           transpose (xs : [t | (h:t) <- xss])

partition               :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs          =  (filter p xs, filter (not . p) xs)

-- group splits its list argument into a list of lists of equal, adjacent
-- elements.  e.g.,
-- group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
group                   :: Eq a => [a] -> [[a]]
group                   =  groupBy (==)

groupBy                 :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy eq []           =  []
groupBy eq (x:xs)       =  (x:ys) : groupBy eq zs
                           where (ys,zs) = span (eq x) xs

-- inits xs returns the list of initial segments of xs, shortest first.
-- e.g., inits "abc" == ["","a","ab","abc"]
inits                   :: [a] -> [[a]]
inits []                =  [[]]
inits (x:xs)            =  [[]] ++ map (x:) (inits xs)

-- tails xs returns the list of all final segments of xs, longest first.
-- e.g., tails "abc" == ["abc", "bc", "c",""]
tails                   :: [a] -> [[a]]
tails []                =  [[]]
tails xxs@(_:xs)        =  xxs : tails xs

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

isSuffixOf              :: Eq a => [a] -> [a] -> Bool
isSuffixOf x y          =  reverse x `isPrefixOf` reverse y

mapAccumL               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumL f s []        =  (s, [])
mapAccumL f s (x:xs)    =  (s'',y:ys)
                           where (s', y ) = f s x
                                 (s'',ys) = mapAccumL f s' xs

mapAccumR               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR f s []        =  (s, [])
mapAccumR f s (x:xs)    =  (s'', y:ys)
                           where (s'',y ) = f s' x
                                 (s', ys) = mapAccumR f s xs
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
unfoldr                 :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b             = case f b of
                                Nothing    -> []
                                Just (a,b) -> a : unfoldr f b

sort                    :: (Ord a) => [a] -> [a]
sort                    =  sortBy compare

sortBy                  :: (a -> a -> Ordering) -> [a] -> [a]
sortBy cmp              =  foldr (insertBy cmp) []

insert                  :: (Ord a) => a -> [a] -> [a]
insert                  = insertBy compare

insertBy                :: (a -> a -> Ordering) -> a -> [a] -> [a]
insertBy cmp x []       =  [x]
insertBy cmp x ys@(y:ys')
                        =  case cmp x y of
                                GT -> y : insertBy cmp x ys'
                                _  -> x : ys

maximumBy               :: (a -> a -> Ordering) -> [a] -> a
maximumBy cmp []        =  error "List.maximumBy: empty list"
maximumBy cmp xs        =  foldl1 max xs
                        where
                           max x y = case cmp x y of
                                        GT -> x
                                        _  -> y

minimumBy               :: (a -> a -> Ordering) -> [a] -> a
minimumBy cmp []        =  error "List.minimumBy: empty list"
minimumBy cmp xs        =  foldl1 min xs
                        where
                           min x y = case cmp x y of
                                        GT -> y
                                        _  -> x

genericLength           :: (Integral a) => [b] -> a
genericLength []        =  0
genericLength (x:xs)    =  1 + genericLength xs

genericTake             :: (Integral a) => a -> [b] -> [b]
genericTake _ []        =  []
genericTake 0 _         =  []
genericTake n (x:xs)
   | n > 0              =  x : genericTake (n-1) xs
   | otherwise          =  error "List.genericTake: negative argument"

genericDrop             :: (Integral a) => a -> [b] -> [b]
genericDrop 0 xs        =  xs
genericDrop _ []        =  []
genericDrop n (_:xs)
   | n > 0              =  genericDrop (n-1) xs
   | otherwise          =  error "List.genericDrop: negative argument"
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
genericSplitAt          :: (Integral a) => a -> [b] -> ([b],[b])
genericSplitAt 0 xs     =  ([],xs)
genericSplitAt _ []     =  ([],[])
genericSplitAt n (x:xs)
   | n > 0              =  (x:xs',xs'')
   | otherwise          =  error "List.genericSplitAt: negative argument"
      where (xs',xs'') =  genericSplitAt (n-1) xs

genericIndex            :: (Integral a) => [b] -> a -> b
genericIndex (x:_)  0   =  x
genericIndex (_:xs) n
        | n > 0         =  genericIndex xs (n-1)
        | otherwise     =  error "List.genericIndex: negative argument"
genericIndex _ _        =  error "List.genericIndex: index too large"

genericReplicate        :: (Integral a) => a -> b -> [b]
genericReplicate n x    =  genericTake n (repeat x)

zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,,)

zip5                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip5                    =  zipWith5 (,,,,)

zip6                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                           [(a,b,c,d,e,f)]
zip6                    =  zipWith6 (,,,,,)

zip7                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                           [g] -> [(a,b,c,d,e,f,g)]
zip7                    =  zipWith7 (,,,,,,)

zipWith4                :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                        =  z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _      =  []

zipWith5                :: (a->b->c->d->e->f) ->
                           [a]->[b]->[c]->[d]->[e]->[f]
zipWith5 z (a:as) (b:bs) (c:cs) (d:ds) (e:es)
                        =  z a b c d e : zipWith5 z as bs cs ds es
zipWith5 _ _ _ _ _ _    =  []

zipWith6                :: (a->b->c->d->e->f->g) ->
                           [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith6 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs)
                        =  z a b c d e f : zipWith6 z as bs cs ds es fs
zipWith6 _ _ _ _ _ _ _  =  []

zipWith7                :: (a->b->c->d->e->f->g->h) ->
                           [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
zipWith7 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs) (g:gs)
                 =  z a b c d e f g : zipWith7 z as bs cs ds es fs gs
zipWith7 _ _ _ _ _ _ _ _ = []

unzip4                  :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip4                  =  foldr (\(a,b,c,d) ~(as,bs,cs,ds) ->
                                    (a:as,b:bs,c:cs,d:ds))
                                 ([],[],[],[])
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press

```
unzip5          :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip5          =  foldr (\(a,b,c,d,e) ~(as,bs,cs,ds,es) ->
                                (a:as,b:bs,c:cs,d:ds,e:es))
                         ([],[],[],[],[])

unzip6          :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip6          =  foldr (\(a,b,c,d,e,f) ~(as,bs,cs,ds,es,fs) ->
                                (a:as,b:bs,c:cs,d:ds,e:es,f:fs))
                         ([],[],[],[],[],[])

unzip7          :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
unzip7          =  foldr (\(a,b,c,d,e,f,g) ~(as,bs,cs,ds,es,fs,gs) ->
                                (a:as,b:bs,c:cs,d:ds,e:es,f:fs,g:gs))
                         ([],[],[],[],[],[],[])
```

https://doi.org/10.1017/S0956796803001916 Published online by Cambridge University Press