

Data.List

Operations on lists.

Basic functions

(++) :: [a] -> [a] -> [a] | [infixr 5](#) | [Source](#)

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] ==
```

If the first list is not finite, the result is the first list.

head :: [a] -> a | [Source](#)

Extract the first element of a list, which must be non-empty.

last :: [a] -> a | [Source](#)

Extract the last element of a list, which must be finite and non-empty.

tail :: [a] -> [a] | [Source](#)

Extract the elements after the head of a list, which must be non-empty.

init :: [a] -> [a] | [Source](#)

Return all the elements of a list except the last one. The list must be non-empty.

uncons :: [a] -> **Maybe** (a, [a]) | [Source](#)

Decompose a list into its head and tail. If the list is empty, returns **Nothing**. If the list is non-empty, returns **Just** (x, xs), where x is the head of the list and xs its tail.

Since 4.8.0.0

Copyright	(c) The University of Glasgow 2001
License	BSD-style (see the file <code>libraries/base/LICENSE</code>)
Maintainer	<code>libraries@haskell.org</code>
Stability	stable
Portability	portable
Safe	Trustworthy
Haskell Language	Haskell2010

Contents

- Basic functions
- List transformations
- Reducing lists (folds)
 - Special folds
- Building lists
 - Scans
 - Accumulating maps
 - Infinite lists
 - Unfolding
- Sublists
 - Extracting sublists
 - Predicates
- Searching lists
 - Searching by equality
 - Searching with a predicate
- Indexing lists
- Zippping and unzipping lists
- Special lists
 - Functions on strings
 - "Set" operations
 - Ordered lists
- Generalized functions
 - The "By" operations
 - User-supplied equality (replacing an `Eq` context)
 - User-supplied comparison (replacing an `Ord` context)
 - The "generic" operations

null :: Foldable t => t a -> Bool

[Source](#)

Test whether the structure is empty. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

length :: Foldable t => t a -> Int

[Source](#)

Returns the size/length of a finite structure as an Int. The default implementation is optimized for structures that are similar to cons-lists, because there is no general way to do better.

List transformations

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

[Source](#)

map f xs is the list obtained by applying f to each element of xs, i.e.,

```
map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
map f [x1, x2, ...] == [f x1, f x2, ...]
```

reverse :: [a] -> [a]

[Source](#)

reverse xs returns the elements of xs in reverse order. xs must be finite.

intersperse :: a -> [a] -> [a]

[Source](#)

The **intersperse** function takes an element and a list and 'intersperses' that element between the elements of the list. For example,

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

intercalate :: [a] -> [[a]] -> [a]

[Source](#)

intercalate xs xss is equivalent to (**concat** (**intersperse** xs xss)). It inserts the list xs in between the lists in xss and concatenates the result.

transpose :: [[a]] -> [[a]]

[Source](#)

The **transpose** function transposes the rows and columns of its argument. For example,

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

If some of the rows are shorter than the following rows, their elements are skipped:

```
transpose [[10,11],[20],[],[30,31,32]] == [[10,20,30],[11,31],[32]]
```

subsequences :: [a] -> [[a]]

[Source](#)

The **subsequences** function returns the list of all subsequences of the argument.

```
subsequences "abc" == ["", "a", "b", "ab", "c", "ac", "bc", "abc"]
```

permutations :: [a] -> [[a]]

[Source](#)

The **permutations** function returns the list of all permutations of the argument.

```
permutations "abc" == ["abc", "bac", "cba", "bca", "cab", "acb"]
```

Reducing lists (folds)

foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b

[Source](#)

Left-associative fold of a structure.

```
foldl f z = foldl f z . toList
```

foldl' :: Foldable t => (b -> a -> b) -> b -> t a -> b

[Source](#)

Left-associative fold of a structure. but with strict application of the operator.

```
foldl f z = foldl' f z . toList
```

foldl1 :: Foldable t => (a -> a -> a) -> t a -> a

[Source](#)

A variant of **foldl** that has no base case, and thus may only be applied to non-empty structures.

```
foldl1 f = foldl1 f . toList
```

foldl1' :: (a -> a -> a) -> [a] -> a

[Source](#)

A strict version of **foldl1**

foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b

[Source](#)

Right-associative fold of a structure.

```
foldr f z = foldr f z . toList
```

foldr1 :: Foldable t => (a -> a -> a) -> t a -> a

[Source](#)

A variant of **foldr** that has no base case, and thus may only be applied to non-empty

structures.

```
foldr1 f = foldr1 f . toList
```

Special folds

```
concat :: Foldable t => t [a] -> [a]
```

[Source](#)

The concatenation of all the elements of a container of lists.

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

[Source](#)

Map a function over all the elements of a container and concatenate the resulting lists.

```
and :: Foldable t => t Bool -> Bool
```

[Source](#)

`and` returns the conjunction of a container of Booleans. For the result to be `True`, the container must be finite; `False`, however, results from a `False` value finitely far from the left end.

```
or :: Foldable t => t Bool -> Bool
```

[Source](#)

`or` returns the disjunction of a container of Booleans. For the result to be `False`, the container must be finite; `True`, however, results from a `True` value finitely far from the left end.

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
```

[Source](#)

Determines whether any element of the structure satisfies the predicate.

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
```

[Source](#)

Determines whether all elements of the structure satisfy the predicate.

```
sum :: (Foldable t, Num a) => t a -> a
```

[Source](#)

The `sum` function computes the sum of the numbers of a structure.

```
product :: (Foldable t, Num a) => t a -> a
```

[Source](#)

The `product` function computes the product of the numbers of a structure.

```
maximum :: forall a. (Foldable t, Ord a) => t a -> a
```

[Source](#)

The largest element of a non-empty structure.

```
minimum :: forall a. (Foldable t, Ord a) => t a -> a
```

[Source](#)

The least element of a non-empty structure.

Building lists

Scans

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

[Source](#)

`scanl` is similar to `foldl`, but returns a list of successive reduced values from the left:

```
scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
```

Note that

```
last (scanl f z xs) == foldl f z xs.
```

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

[Source](#)

A strictly accumulating version of `scanl`

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

[Source](#)

`scanl1` is a variant of `scanl` that has no starting value argument:

```
scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]
```

```
scanr :: (a -> b -> b) -> b -> [a] -> [b]
```

[Source](#)

`scanr` is the right-to-left dual of `scanl`. Note that

```
head (scanr f z xs) == foldr f z xs.
```

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

[Source](#)

`scanr1` is a variant of `scanr` that has no starting value argument.

Accumulating maps

```
mapAccumL :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
```

[Source](#)

The `mapAccumL` function behaves like a combination of `fmap` and `foldl`; it applies a function to each element of a structure, passing an accumulating parameter from left to right, and returning a final value of this accumulator together with the new structure.

```
mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
```

[Source](#)

The `mapAccumR` function behaves like a combination of `fmap` and `foldr`; it applies a function to each element of a structure, passing an accumulating parameter from right to left, and returning a final value of this accumulator together with the new structure.

Infinite lists

```
iterate :: (a -> a) -> a -> [a]
```

[Source](#)

`iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x == [x, f x, f (f x), ...]
```

```
repeat :: a -> [a]
```

[Source](#)

`repeat x` is an infinite list, with `x` the value of every element.

```
replicate :: Int -> a -> [a]
```

[Source](#)

`replicate n x` is a list of length `n` with `x` the value of every element. It is an instance of the more general `genericReplicate`, in which `n` may be of any integral type.

```
cycle :: [a] -> [a]
```

[Source](#)

`cycle` ties a finite list into a circular one, or equivalently, the infinite repetition of the original list. It is the identity on infinite lists.

Unfolding

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

[Source](#)

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. The function takes the element and returns `Nothing` if it is done producing the list or returns `Just (a,b)`, in which case, `a` is prepended to the list and `b` is used as the next element in a recursive call. For example,

```
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
```

A simple use of unfoldr:

```
unfoldr (\b -> if b == 0 then Nothing else Just (b, b-1)) 10
[10,9,8,7,6,5,4,3,2,1]
```

Sublists

Extracting sublists

```
take :: Int -> [a] -> [a]
```

[Source](#)

take n, applied to a list xs, returns the prefix of xs of length n, or xs itself if n > **length** xs:

```
take 5 "Hello World!" == "Hello"
take 3 [1,2,3,4,5] == [1,2,3]
take 3 [1,2] == [1,2]
take 3 [] == []
take (-1) [1,2] == []
take 0 [1,2] == []
```

It is an instance of the more general **genericTake**, in which n may be of any integral type.

```
drop :: Int -> [a] -> [a]
```

[Source](#)

drop n xs returns the suffix of xs after the first n elements, or [] if n > **length** xs:

```
drop 6 "Hello World!" == "World!"
drop 3 [1,2,3,4,5] == [4,5]
drop 3 [1,2] == []
drop 3 [] == []
drop (-1) [1,2] == [1,2]
drop 0 [1,2] == [1,2]
```

It is an instance of the more general **genericDrop**, in which n may be of any integral type.

```
splitAt :: Int -> [a] -> ([a], [a])
```

[Source](#)

splitAt n xs returns a tuple where first element is xs prefix of length n and second element is the remainder of the list:

```
splitAt 6 "Hello World!" == ("Hello ", "World!")
splitAt 3 [1,2,3,4,5] == ([1,2,3], [4,5])
splitAt 1 [1,2,3] == ([1], [2,3])
```

```
splitAt 3 [1,2,3] == ([1,2,3],[])
splitAt 4 [1,2,3] == ([1,2,3],[])
splitAt 0 [1,2,3] == ([],[1,2,3])
splitAt (-1) [1,2,3] == ([],[1,2,3])
```

It is equivalent to (`take n xs`, `drop n xs`) when `n` is not `_|_` (`splitAt _|_ xs = _|_`). `splitAt` is an instance of the more general `genericSplitAt`, in which `n` may be of any integral type.

takeWhile :: (a -> Bool) -> [a] -> [a]

[Source](#)

`takeWhile`, applied to a predicate `p` and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy `p`:

```
takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
takeWhile (< 9) [1,2,3] == [1,2,3]
takeWhile (< 0) [1,2,3] == []
```

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

[Source](#)

`dropWhile p xs` returns the suffix remaining after `takeWhile p xs`:

```
dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3]
dropWhile (< 9) [1,2,3] == []
dropWhile (< 0) [1,2,3] == [1,2,3]
```

dropWhileEnd :: (a -> Bool) -> [a] -> [a]

[Source](#)

The `dropWhileEnd` function drops the largest suffix of a list in which the given predicate holds for all elements. For example:

```
dropWhileEnd isSpace "foo\n" == "foo"
dropWhileEnd isSpace "foo bar" == "foo bar"
dropWhileEnd isSpace ("foo\n" ++ undefined) == "foo" ++ undefined
```

Since: 4.5.0.0

span :: (a -> Bool) -> [a] -> ([a], [a])

[Source](#)

`span`, applied to a predicate `p` and a list `xs`, returns a tuple where first element is longest prefix (possibly empty) of `xs` of elements that satisfy `p` and second element is the remainder of the list:

```
span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4])
span (< 9) [1,2,3] == ([1,2,3],[])
span (< 0) [1,2,3] == ([],[1,2,3])
```

`span p xs` is equivalent to (`takeWhile p xs`, `dropWhile p xs`)

break :: (a -> Bool) -> [a] -> ([a], [a])

[Source](#)

break, applied to a predicate *p* and a list *xs*, returns a tuple where first element is longest prefix (possibly empty) of *xs* of elements that *do not satisfy* *p* and second element is the remainder of the list:

```
break (> 3) [1,2,3,4,1,2,3,4] == ([1,2,3],[4,1,2,3,4])
break (< 9) [1,2,3] == ([],[1,2,3])
break (> 9) [1,2,3] == ([1,2,3],[])
```

break *p* is equivalent to **span** (**not** . *p*).

stripPrefix :: Eq a => [a] -> [a] -> Maybe [a]

[Source](#)

The **stripPrefix** function drops the given prefix from a list. It returns **Nothing** if the list did not start with the prefix given, or **Just** the list after the prefix, if it does.

```
stripPrefix "foo" "foobar" == Just "bar"
stripPrefix "foo" "foo" == Just ""
stripPrefix "foo" "barfoo" == Nothing
stripPrefix "foo" "barfoobaz" == Nothing
```

group :: Eq a => [a] -> [[a]]

[Source](#)

The **group** function takes a list and returns a list of lists such that the concatenation of the result is equal to the argument. Moreover, each sublist in the result contains only equal elements. For example,

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

It is a special case of **groupBy**, which allows the programmer to supply their own equality test.

inits :: [a] -> [[a]]

[Source](#)

The **inits** function returns all initial segments of the argument, shortest first. For example,

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

Note that **inits** has the following strictness property: **inits** (xs ++ _|_) = **inits** xs ++ _|_

In particular, **inits** _|_ = [] : _|_

tails :: [a] -> [[a]]

[Source](#)

The **tails** function returns all final segments of the argument, longest first. For example,

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

Note that `tails` has the following strictness property: `tails _|_ = _|_ : _|_`

Predicates

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
```

 | Source

The `isPrefixOf` function takes two lists and returns `True` iff the first list is a prefix of the second.

```
isSuffixOf :: Eq a => [a] -> [a] -> Bool
```

 | Source

The `isSuffixOf` function takes two lists and returns `True` iff the first list is a suffix of the second. The second list must be finite.

```
isInfixOf :: Eq a => [a] -> [a] -> Bool
```

 | Source

The `isInfixOf` function takes two lists and returns `True` iff the first list is contained, wholly and intact, anywhere within the second.

Example:

```
isInfixOf "Haskell" "I really like Haskell." == True
isInfixOf "Ial" "I really like Haskell." == False
```

```
isSubsequenceOf :: Eq a => [a] -> [a] -> Bool
```

 | Source

The `isSubsequenceOf` function takes two lists and returns `True` if the first list is a subsequence of the second list.

`isSubsequenceOf x y` is equivalent to `elem x (subsequences y)`.

Examples

Since: 4.8.0.0

Searching lists

Searching by equality

```
elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

 | Source

Does the element occur in the structure?

```
notElem :: (Foldable t, Eq a) => a -> t a -> Bool
```

 | infix 4 | Source

`notElem` is the negation of `elem`.

lookup :: `Eq a => a -> [(a, b)] -> Maybe b` | Source

`lookup` key `assocs` looks up a key in an association list.

Searching with a predicate

find :: `Foldable t => (a -> Bool) -> t a -> Maybe a` | Source

The `find` function takes a predicate and a structure and returns the leftmost element of the structure matching the predicate, or `Nothing` if there is no such element.

filter :: `(a -> Bool) -> [a] -> [a]` | Source

`filter`, applied to a predicate and a list, returns the list of those elements that satisfy the predicate; i.e.,

`filter p xs = [x | x <- xs, p x]`

partition :: `(a -> Bool) -> [a] -> ([a], [a])` | Source

The `partition` function takes a predicate a list and returns the pair of lists of elements which do and do not satisfy the predicate, respectively; i.e.,

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

Indexing lists

These functions treat a list `xs` as a indexed collection, with indices ranging from 0 to `length xs - 1`.

(!!) :: `[a] -> Int -> a` | infixl 9 | Source

List index (subscript) operator, starting from 0. It is an instance of the more general `genericIndex`, which takes an index of any integral type.

elemIndex :: `Eq a => a -> [a] -> Maybe Int` | Source

The `elemIndex` function returns the index of the first element in the given list which is equal (by `==`) to the query element, or `Nothing` if there is no such element.

elemIndices :: `Eq a => a -> [a] -> [Int]` | Source

The `elemIndices` function extends `elemIndex`, by returning the indices of all elements

equal to the query element, in ascending order.

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
```

[Source](#)

The `findIndex` function takes a predicate and a list and returns the index of the first element in the list satisfying the predicate, or `Nothing` if there is no such element.

```
findIndices :: (a -> Bool) -> [a] -> [Int]
```

[Source](#)

The `findIndices` function extends `findIndex`, by returning the indices of all elements satisfying the predicate, in ascending order.

Zippping and unzipping lists

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

[Source](#)

`zip` takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.

`zip` is right-lazy:

```
zip [] _ = []
```

```
zip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
```

[Source](#)

`zip3` takes three lists and returns a list of triples, analogous to `zip`.

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

[Source](#)

The `zip4` function takes four lists and returns a list of quadruples, analogous to `zip`.

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

[Source](#)

The `zip5` function takes five lists and returns a list of five-tuples, analogous to `zip`.

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

[Source](#)

The `zip6` function takes six lists and returns a list of six-tuples, analogous to `zip`.

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

[Source](#)

The `zip7` function takes seven lists and returns a list of seven-tuples, analogous to `zip`.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

[Source](#)

`zipWith` generalises `zip` by zipping with the function given as the first argument, instead of a tupling function. For example, `zipWith (+)` is applied to two lists to produce the list of corresponding sums.

`zipWith` is right-lazy:

```
zipWith f [] _ = []
```

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
```

[Source](#)

The `zipWith3` function takes a function which combines three elements, as well as three lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith4 :: (a -> b -> c -> d -> e) -> [a] -> [b] -> [c] -> [d] -> [e]
```

[Source](#)

The `zipWith4` function takes a function which combines four elements, as well as four lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith5 :: (a -> b -> c -> d -> e -> f) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f]
```

[Source](#)

The `zipWith5` function takes a function which combines five elements, as well as five lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith6 :: (a -> b -> c -> d -> e -> f -> g) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
```

[Source](#)

The `zipWith6` function takes a function which combines six elements, as well as six lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
zipWith7 :: (a -> b -> c -> d -> e -> f -> g -> h) -> [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g] -> [h]
```

[Source](#)

The `zipWith7` function takes a function which combines seven elements, as well as seven lists and returns a list of their point-wise combination, analogous to `zipWith`.

```
unzip :: [(a, b)] -> ([a], [b])
```

[Source](#)

`unzip` transforms a list of pairs into a list of first components and a list of second components.

```
unzip3 :: [(a, b, c)] -> ([a], [b], [c])
```

[Source](#)

The `unzip3` function takes a list of triples and returns three lists, analogous to `unzip`.

unzip4 :: [(a, b, c, d)] -> ([a], [b], [c], [d]) | Source

The **unzip4** function takes a list of quadruples and returns four lists, analogous to **unzip**.

unzip5 :: [(a, b, c, d, e)] -> ([a], [b], [c], [d], [e]) | Source

The **unzip5** function takes a list of five-tuples and returns five lists, analogous to **unzip**.

unzip6 :: [(a, b, c, d, e, f)] -> ([a], [b], [c], [d], [e], [f]) | Source

The **unzip6** function takes a list of six-tuples and returns six lists, analogous to **unzip**.

unzip7 :: [(a, b, c, d, e, f, g)] -> ([a], [b], [c], [d], [e], [f], [g]) | Source

The **unzip7** function takes a list of seven-tuples and returns seven lists, analogous to **unzip**.

Special lists

Functions on strings

lines :: String -> [String] | Source

lines breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

words :: String -> [String] | Source

words breaks a string up into a list of words, which were delimited by white space.

unlines :: [String] -> String | Source

unlines is an inverse operation to **lines**. It joins lines, after appending a terminating newline to each.

unwords :: [String] -> String | Source

unwords is an inverse operation to **words**. It joins words with separating spaces.

"Set" operations

nub :: Eq a => [a] -> [a] | Source

$O(n^2)$. The **nub** function removes duplicate elements from a list. In particular, it keeps

only the first occurrence of each element. (The name **nub** means `essence'.) It is a special case of **nubBy**, which allows the programmer to supply their own equality test.

delete :: Eq a => a -> [a] -> [a]	Source
---	------------------------

delete x removes the first occurrence of x from its list argument. For example,

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

It is a special case of **deleteBy**, which allows the programmer to supply their own equality test.

(\\) :: Eq a => [a] -> [a] -> [a]	infix 5	Source
---	---------	------------------------

The **** function 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.
```

It is a special case of **deleteFirstBy**, which allows the programmer to supply their own equality test.

union :: Eq a => [a] -> [a] -> [a]	Source
--	------------------------

The **union** function returns the list union of the two lists. For example,

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

Duplicates, and elements of the first list, are removed from the the second list, but if the first list contains duplicates, so will the result. It is a special case of **unionBy**, which allows the programmer to supply their own equality test.

intersect :: Eq a => [a] -> [a] -> [a]	Source
--	------------------------

The **intersect** function takes the list intersection of two lists. For example,

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

If the first list contains duplicates, so will the result.

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

It is a special case of **intersectBy**, which allows the programmer to supply their own equality test. If the element is found in both the first and the second list, the element from the first list will be used.

Ordered lists

```
sort :: Ord a => [a] -> [a]
```

[Source](#)

The `sort` function implements a stable sorting algorithm. It is a special case of `sortBy`, which allows the programmer to supply their own comparison function.

```
sortOn :: Ord b => (a -> b) -> [a] -> [a]
```

[Source](#)

Sort a list by comparing the results of a key function applied to each element. `sortOn f` is equivalent to `sortBy . comparing f`, but has the performance advantage of only evaluating `f` once for each element in the input list. This is called the decorate-sort-undecorate paradigm, or Schwartzian transform.

Since: 4.8.0.0

```
insert :: Ord a => a -> [a] -> [a]
```

[Source](#)

The `insert` function takes an element and a list and inserts the element into the list at the first position where it is less than or equal to the next element. In particular, if the list is sorted before the call, the result will also be sorted. It is a special case of `insertBy`, which allows the programmer to supply their own comparison function.

Generalized functions

The "By" operations

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with ``By'`.

It is often convenient to use these functions together with `on`, for instance `sortBy (compare `on` fst)`.

User-supplied equality (replacing an Eq context)

The predicate is assumed to define an equivalence.

```
nubBy :: (a -> a -> Bool) -> [a] -> [a]
```

[Source](#)

The `nubBy` function behaves just like `nub`, except it uses a user-supplied equality predicate instead of the overloaded `==` function.

```
deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]
```

[Source](#)

The `deleteBy` function behaves like `delete`, but takes a user-supplied equality predicate.

```
deleteFirstsBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]
```

[Source](#)

The `deleteFirstsBy` function takes a predicate and two lists and returns the first list with the first occurrence of each element of the second list removed.

`unionBy` :: (a -> a -> Bool) -> [a] -> [a] -> [a] | Source

The `unionBy` function is the non-overloaded version of `union`.

`intersectBy` :: (a -> a -> Bool) -> [a] -> [a] -> [a] | Source

The `intersectBy` function is the non-overloaded version of `intersect`.

`groupBy` :: (a -> a -> Bool) -> [a] -> [[a]] | Source

The `groupBy` function is the non-overloaded version of `group`.

User-supplied comparison (replacing an Ord context)

The function is assumed to define a total ordering.

`sortBy` :: (a -> a -> Ordering) -> [a] -> [a] | Source

The `sortBy` function is the non-overloaded version of `sort`.

`insertBy` :: (a -> a -> Ordering) -> a -> [a] -> [a] | Source

The non-overloaded version of `insert`.

`maximumBy` :: Foldable t => (a -> a -> Ordering) -> t a -> a | Source

The largest element of a non-empty structure with respect to the given comparison function.

`minimumBy` :: Foldable t => (a -> a -> Ordering) -> t a -> a | Source

The least element of a non-empty structure with respect to the given comparison function.

The "generic" operations

The prefix ``generic'` indicates an overloaded function that is a generalized version of a `Prelude` function.

`genericLength` :: Num i => [a] -> i | Source

The `genericLength` function is an overloaded version of `length`. In particular, instead of

returning an `Int`, it returns any type which is an instance of `Num`. It is, however, less efficient than `length`.

`genericTake` :: `Integral i => i -> [a] -> [a]` | [Source](#)

The `genericTake` function is an overloaded version of `take`, which accepts any `Integral` value as the number of elements to take.

`genericDrop` :: `Integral i => i -> [a] -> [a]` | [Source](#)

The `genericDrop` function is an overloaded version of `drop`, which accepts any `Integral` value as the number of elements to drop.

`genericSplitAt` :: `Integral i => i -> [a] -> ([a], [a])` | [Source](#)

The `genericSplitAt` function is an overloaded version of `splitAt`, which accepts any `Integral` value as the position at which to split.

`genericIndex` :: `Integral i => [a] -> i -> a` | [Source](#)

The `genericIndex` function is an overloaded version of `!!`, which accepts any `Integral` value as the index.

`genericReplicate` :: `Integral i => i -> a -> [a]` | [Source](#)

The `genericReplicate` function is an overloaded version of `replicate`, which accepts any `Integral` value as the number of repetitions to make.