

zip

A prelude function that's often used in list comprehensions is

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

We'll look at zip's definition soon; for now, let an example suffice to show what zip does:

```
Main> zip [0..5] "The Merry Month of September"
[(0,'T'),(1,'h'),(2,'e'),(3,' '),(4,'M'),(5,'e')]
```

Example: computing the scalar (dot) product of two vectors

$$a \cdot b = \sum_i a_i \times b_i$$

where the vectors are represented by lists.

If we traverse each list with its own generator...

```
> dot as bs = sum [ a*b | a <- as, b <- bs ]
```

then we get

```
dot [1,2,3] [4,5,6]
  ~> sum [1*4,1*5,1*6,2*4,2*5,2*6,3*4,3*5,3*6]
```

whereas what we want is

```
sum [1*4, 2*5, 3*6]
```

Using zip enables both lists to be traversed in step, using a single generator:

```
> dot as bs = sum [ a*b | (a,b) <- zip as bs ]
```

Note the pattern (a,b), which *names* the elements of each pair produced by zip.

Prelude list-processing functions

In addition to arithmetic sequences and list comprehensions, there is a large collection of list-processing functions in the Prelude, including

```
(++) :: [a] -> [a] -> [a]
Concatenates two lists.
"hippo" ++ "drome" ~> "hippodrome"

concat :: [[a]] -> [a]
Concatenates a list of lists into a single list.
concat ["Cat", "Mouse", "Duck"]
  ~> "CatMouseDuck"

length :: [a] -> Int
Counts a list's elements.
length ["Cat", "Mouse", "Duck"] ~> 3

head/last :: [a] -> a
Returns a list's first/last element.
head "Duck" ~> 'D'
last "Duck" ~> 'k'

tail/init :: [a] -> [a]
Returns all of a list except its first/last element.
tail "Duck" ~> "uck"
init "Duck" ~> "Duc"

replicate :: Int -> a -> [a]
Makes a list of n copies of an item.
replicate 3 "Cat" ~> ["Cat", "Cat", "Cat"]
```

```
take :: Int -> [a] -> [a]
take n returns a list's first n elements.
take 3 "Mouse" ~> "Mou"

drop :: Int -> [a] -> [a]
drop n returns all except a list's first n elements.
drop 3 "Mouse" ~> "se"

splitAt :: Int -> [a] -> ([a],[a])
Splits a list at a given position.
splitAt 3 "Mouse" ~> ("Mou", "se")

takeWhile :: (a -> Bool) -> [a] -> [a]
Returns elements of a list up to the first that fails a given test.
takeWhile (<4) [0,2,1,5,3] ~> [0,2,1]

dropWhile :: (a -> Bool) -> [a] -> [a]
Returns elements of a list beginning with the first that fails a given test.
dropWhile (<4) [0,2,1,5,3] ~> [5,3]

reverse :: [a] -> [a]
Reverses the order of a list's elements.
reverse "Mouse" ~> "esuoM"

null :: [a] -> Bool
Empty-list test.
  ☞ Better than (==[]) and (==0).length
```

```
zip :: [a] -> [b] -> [(a,b)]
Maps two lists into a list of pairs.
zip "Cat" "Mouse"
  ~> [('C','M'), ('a','o'), ('t','u')]

unzip :: [(a,b)] -> ([a],[b])
Maps a list of pairs into a pair of lists.
unzip [('C','M'), ('a','o'), ('t','u')]
  ~> ("Cat", "Mou")

and, or :: [Bool] -> Bool
Returns a Boolean list's conjunction/disjunction.
and [3<0, 0<3, 3>=0] ~> False
or [3>0, 0<3, 3>=0] ~> True

sum :: Num a => [a] -> a
Returns a numeric list's sum.
sum [1..4] ~> 10

product :: Num a => [a] -> a
Returns a numeric list's product.
product [1..4] ~> 24

map, filter (defined earlier)

elem :: Eq a => a -> [a] -> Bool
Membership test.
3 `elem` [0..5] ~> True
```

Useful though all these functions are, eventually one is bound to encounter a list-processing problem for which there is no predefined function.

This requires the use of ...

The List Constructor (:)

To build a list one element at a time, use the operator

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

which is pronounced “*followed-by*” (in Lisp, it’s *cons*).

In an expression, (:) constructs a list from an item and a list.

The left argument of (:) is *an item* of some type; its right argument is *a list of items* of the same type.

Some equivalences:

```
[1,2,3] = 1:(2:(3:[]))
```

```
"123" = '1':('2':('3':[]))
```

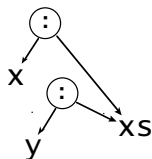
```
[1,2,3] = 1:[2,3]
```

```
x:xs = [x] ++ xs
```

Note: (:) is *right-associative*, so

```
1:(2:(3:[])) = 1:2:3:[]
```

Given a list *xs*, creating lists *x:xs* and *y:xs* requires no copying—the tail *xs* is **shared**:



Because (:) is defined to be *non-strict*,

```
[n..] = let f k = k : f (k+1) in f n
```

The list *[n..]* is *infinite*, but thanks to nonstrictness it occupies only finite space; each element is constructed only on demand.

...and the List Selector (:)

The *same operator* is also used in *patterns*, where it serves as a selector. Examples:

```
> head :: [a] -> a
```

```
> head (x:_) = x -- matches non-empty lists
```

```
> last :: [a] -> a
```

```
> last [x] = x -- matches 1-element lists ONLY
```

```
> last (_:xs) = last xs
```

```
> tail :: [a] -> [a]
```

```
> tail (_:xs) = xs
```

```
> init :: [a] -> [a]
```

```
> init [x] = []
```

```
> init (x:xs) = x : init xs
```

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

```
> length [] = 0
```

```
> length (x:xs) = 1 + length xs
```

```
> reverse :: [a] -> [a]
```

```
> reverse [] = []
```

```
> reverse (x:xs) = reverse xs ++ [x]
```

Names such as *x* and *xs* (or *b* and *bs*, etc.), are commonly used in list patterns *x:xs* (or *b:bs*) as a reminder that

- *x* (or *b*, etc.) names the matching list’s *first element*,
- *xs* (or *bs*, etc.) names its *remaining elements*.

Another example— the list-indexing operator:

```
> (!!) :: [a] -> Int -> a
```

```
> (x:xs) !! 0 = x
```

```
> (x:xs) !! n | n > 0 = xs !! (n-1)
```

```
> (_:_) !! _ = error "negative index"
```

```
> [] !! _ = error "index too large"
```

List indexing is *expensive* (and used too often by students still thinking in Java).

Using (:) as both selector and constructor, we can define *zip*:

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

```
> zip [] = []
```

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

```
> zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

... and concatenation:

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

```
> [] ++ ys = ys
```

```
> (x:xs) ++ ys = x : (xs ++ ys)
```

```
> -- as ++ bs includes bs, which is not copied
```

... and functions such as

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

```
> take n _ | n <= 0 = []
```

```
> take _ [] = []
```

```
> take n (x:xs) = x : take (n-1) xs
```

```
> drop n xs | n <= 0 = xs
```

```
> drop _ [] = []
```

```
> drop n (_:xs) = drop (n-1) xs
```

which are used to select parts of lists.

The empty-list test:

```
null :: [a] -> Bool
```

```
null [] = True
```

```
null _ = False
```

Explaining *list comprehensions* in terms of (:):

```
[ f y | y <- (x:xs) ]
```

```
~> f x : [ f y | y <- xs ]
```

Note that *xs = tail(x:xs)*

```
[ f y | y <- [] ]
```

```
~> []
```

```
[ f y | y <- (x:xs), g y ]
```

```
~> if g x
```

```
then f x : [ f y | y <- xs, g y ]
```

```
else [ f y | y <- xs, g y ]
```

```
[ f y | y <- [], g y ]
```

```
~> []
```