Contents

Function

head

`head xs` returns the first element of `xs`, if non-empty

- Type Signature: `head :: [a] -> a`
- Definition

```
head (x:_) = x
head [] = error "Prelude.head: empty list"
```

tail

`tail xs`, for non-empty `xs` returns it with first element removed

- Type Signature: `tail :: [a] -> [a]`
- Definition

```
tail (_:xs) = xs
tail [] = error "Prelude.tail: empty list"
```

tail [] /= [] - Why Not?

Why don't we define `tail [] = []`? The typing allows it

If we have a special `headInt :: [Int] -> Int`, why wouldn't we define `headInt [] = 0`

A key design principle behind Haskell libraries and programs is to have programs (functions!) that obey nice obvious laws

```
xs = head xs ++ tail xs
sum (xs ++ ys) = sum xs + sum ys
prod (xs ++ ys) = prod xs * prod ys
```

Imagine if we defined

```
prod xs = headInt xs * prod (tail xs)
```

last

`last xs` returns the last element of `xs` if non-empty

- Type Signature: `last :: [a] -> a`
- Definition

```
last[x] = x
last (x:xs) = last xs
last [] = error "Prelude.last: empty list
```

init

`init xs` for non-empty `xs` returns it with last element removed

- Type signature: `init :: [a] -> a`
- Definitions

```
init [x] = []
init (x:xs) = x : init xs
init [] = error "Prelude.init: empty list"
```

null

`null xs` returns `True` is the list is empty

- Type Signature: `null :: [a] -> Bool`
- Definitions

```
null [] = True
null (_:_) = False
```

(!!)

`(!!) xs n`, or `xs !! n` selects the `n`th element of list `xs`, provided it is long enough. Indices start at 0.

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

- Fixity: `infixl 9 !!`

- Definitions

```
xs !! n | n < 0 = error "Prelude.!!: negative index"
[] !! _ = error "Prelude.!!: index too large"
(x:_) !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

++

`xs ++ ys` joins list `xs` and `ys` together

- Type Signature: `(++) :: [a] -> [a] -> [a]`
- Definitions

```
[] ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

reverse

- `reservse xs`, reverses list `xs`
- Type Signature: `reverse :: [a] -> [a]`
- Definitions

```
reverse xs = rev [] xs`
    where
        rev sx (x:xs) = rev (x:sx) xs
        rev sx []     = sx
```

Prelude Version

- Definitions

```
reverse = foldl (flip(:)) []
```

- Prelude doesn't always give the most obvious definition of a function's behaviour