λ Lessons

Pattern matching, first-class functions, and abstracting over recursion in Haskell

This is a short, interactive lesson that teaches core functional programming concepts. It was designed to transform the way you think about performing operations on lists of things, by **showing you how functions are executed.**

You can explore the way `map` and `fold` (`foldr` and `foldl`) are defined and computed. Feel free to **re-define** any of the functions used in this document in the **Function Editor**.

This document implements a small, dynamically-typed, subset of Haskell that includes integers, lists, functions, pattern matching and recursion.

Built by Jan Paul Posma & Steve Krouse at YC Hacks '14 with React.js & PEG.js. Inspired by Bret Victor & Brent Yorgey. Check out the source.

Function Editor

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

foldr :: (a -> b -> b
foldr f i []     = i
foldr f i (x:xs) = f

foldl :: (a -> b -> a
foldl f i []     = i
foldl f i (x:xs) = fo

cons :: Int -> [Int]
cons x xs = x : xs

reverseCons :: [Int]
reverseCons xs x = x

plus :: Int -> Int ->
plus x y = x + y

addOne :: Int -> Int
addOne x = x + 1

double :: Int -> Int
double x = x + x

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

`map`

`map` is a function that performs some operation on every element in a list.

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

`map` takes 2 inputs

- function of type `(a -> b)`
- list of type `[a]`

and returns

- list of type `[b]`

The base-case of `map` pattern matches on `[]` and returns `[]`.

The recursive-case of `map` pattern matches on the first list element `x` and returns `(f x) : map f xs`.

```
(map addOne [1,2,3,4,5])  (edit)  (clear)
```
↑ click to expand execution

Function Editor

fold

`fold` describes 2 functions that "summarize" the elements in a list.

- `foldr` - "fold right", applies `f` to `x` and the result of folding `f` over the rest (remember: `foldr` moves to the right as it computes with the computation on the outside)
- `foldl` - "fold left", evaluates `f x i` immediately and uses that as the new initial value for folding `f` over the rest (remember: `foldl` stays on the left as it computes with the computation on the inside)

foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)
```

`foldr` takes 3 inputs

- function of type `(a -> b -> b)`
- initial value of type `b`
- list of type `[a]`

and returns

- accumulated value of type `b`

The base-case of `foldr` pattern matches on `[]` and returns `i`.

The recursive-case of `foldr` pattern matches on the first list element `x` and returns `f x (foldr f i xs)`.

```
(foldr plus 0 [1,2,3,4,5]) (edit) (clear)
```

↑ click to expand execution

foldl

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f i [] = i
foldl f i (x:xs) = foldl f (f i x) xs
```

`foldl` takes 3 inputs

- function of type `(a -> b -> a)`
- initial value of type `a`
- list of type `[b]`

and returns

- accumulated value of type `a`

The base-case of `foldl` pattern matches on `[]` and returns `i`.

The recursive-case of `foldl` pattern matches on the first list element `x` and returns `foldl f (f i x) xs`.

Function Editor

```
(foldl reverseCons [] [1,2,3,4,5])  (edit)  (clear)
```

↑ click to expand execution

Function Editor