A start on higher types: Mapping, 1

Mapping via list comprehension

```haskell
doubleAll :: [Int] -> [Int]
doubleAll lst = [2*x | x <- lst]

addPairs :: [(Int,Int)] -> [[Int]]
addPairs mns = [[m+n] | (m,n) <- mns]

multAll :: Int -> [Int] -> [Int]
multAll x ys = [x*y | y <- ys]
```

More generally for any function `f :: a -> b`, we can define a function

```haskell
apply_f :: [a] -> [b]
apply_f xs = [f x | x <- xs]
```

A start on higher types: Mapping, 2

Mapping via structural recursion over lists

```haskell
doubleAll' :: [Int] -> [Int]
doubleAll' [] = []
doubleAll' (x:xs) = (2*x):doubleAll xs

addPairs' :: [(Int,Int)] -> [[Int]]
addPairs' [] = []
addPairs' ((m,n):mns) = [m+n]:addPairs mns

multAll' :: Int -> [Int] -> [Int]
multAll' x [] = []
multAll' x (y:ys) = (x*y):(multAll' x ys)
```

More generally for any function `f :: a -> b`, we can define a function

```haskell
apply_f' :: [a] -> [b]
apply_f' [] = []
apply_f' (x:xs) = (f x):apply_f' xs
```
A start on higher types: Mapping, 3

Mapping via map

Let us define a generic function to do mapping:

```haskell
map :: (a -> b) -> [a] -> [b]
map f lst = [ f x | x <- lst ]

— or —

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

map is higher order, it accepts a function as an argument. E.g.,

- `map fst [(1,False), (3,True), (-5,False), (34,False)]`;
  
  `[1,3,-5,34]`

- `map length [[1,5,6], [3,5], [], [3..10]]`
  
  `[3,2,0,8]`

- `map sum [[1,5,6], [3,5], [], [3..10]]`
  
  `[12,8,0,52]`

A start on higher types: Filtering, 1

Filtering elements from a list via list comprehensions

```haskell
lessThan10 :: [Int] -> [Int]
lessThan10 xs = [ x | x <- xs, x<10 ]

offDiagonal :: [(Int,Int)] -> [(Int,Int)]
offDiagonal mns = [(m,n) | (m,n) <- mns , m/=n]
```

functions as first-class values

In functional languages (generally), functions are *first-class values*, i.e. are treated just like any other value.

So functions can be

- passed as arguments to functions
- returned as results from functions
- bound to variables
- expressed without being given a name (\(\lambda\)-expressions)
- elements of list (and other data structures)
- ...

A function that

- (i) accepts functions as arguments or
- (ii) returns a function as a value or
- (iii) both (i) and (ii)

is higher order. E.g., `map` and `filter`.

A start on higher types: Filtering, 2

Here is a generic way of doing filtering:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p lst = [ x | x <- lst, p x ]

— or —

filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs) | p x = x:(filter' p xs)
| otherwise = filter' p xs
```

So

```haskell
isOffDiag :: (Int,Int) -> Bool
isOffDiag (m,n) = (m/=n)

filter isOffDiag [(3,4),(5,5),(10,-2),(99,99)] ~ [(3,4),(10,-2)]

filter isDigit "a37b29?" ~ "379?"

filter not [True,False,False,True] ~ [False,False]
```
**Higher-type goodies, 1**

\[
\text{dropWhile, takeWhile} \\
\quad :: (a -> \text{Bool}) \rightarrow [a] \rightarrow [a]
\]

\[
\text{dropWhile } p \ [\] = [] \\
\text{dropWhile } p \ (x:xs) \quad | \quad p \ x = \text{dropWhile } p \ xs \\
| \quad \text{otherwise} = x:xs
\]

\[
\text{takeWhile } p \ [\] = [] \\
\text{takeWhile } p \ (x:xs) \quad | \quad p \ x = x : \text{takeWhile } p \ xs \\
| \quad \text{otherwise} = []
\]

**Q:** What is \(<10\) doing?

**Q:** What is “.” doing??

**For example:**

\[
\text{takeWhile } (<10) [0,3..20] \quad \sim \quad [0,3,6,9] \\
\text{dropWhile } (<10) [0,3..20] \quad \sim \quad [12,15,18] \\
\text{dropWhile } \text{isSpace} \ " \text{hi there} " \quad \sim \quad \" \text{hi there} \" \\
\text{takeWhile } \text{not . isSpace} \ "\text{hi there} \" \quad \sim \quad \" \text{hi} \" \\
\text{dropWhile } \text{not . isSpace} \ "\text{hi there} \" \quad \sim \quad \" \text{there} \"
\]

**Digression: Sections and the composition operator**

**Sections**

\[
\begin{align*}
10 + 3 & \equiv (+)\ 10\ 3 \\
10 \equiv 3 & \equiv (==)\ 10\ 3 \\
10 \times 3 & \equiv (\times)\ 10\ 3 \\
10 \div 3 & \equiv (\div)\ 10\ 3
\end{align*}
\]

\[
(\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c
\]

\[
(f \ .\ g)\ x = f\ (g\ x)
\]

**Example:** Define a function \(\text{trim}\) that deletes leading and trailing white space from a string

\[
\text{trimFront } \text{str} = \text{dropWhile } \text{isSpace} \ \text{str} \\
\text{trim } \text{str} = \text{reverse } (\text{trimFront } (\text{reverse } (\text{trimFront } \text{str}))) \\
\quad \quad \quad \quad \quad \text{or better yet} \\
\text{trim'} = \text{reverse } .\ \text{trimFront } .\ \text{reverse } .\ \text{trimFront}
\]

**Higher-type goodies, 2**

\[
\text{span } :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a],[a])
\]

\[
\text{span } p \ [\] = ([],[]) \\
\text{span } p\ \text{x@}(x:xs') \quad | \quad p\ x = (x:ys,zs) \\
| \quad \text{otherwise} = ([],xs) \\
\quad \text{where } (ys,zs) = \text{span } p\ xs'
\]

**For example:**

\[
\text{span } (<10) [0,3..20] \quad \sim \quad ([0,3,6,9],[12,15,18]) \\
\text{span } \text{isSpace} \ "\text{hi there} \" \quad \sim \quad (\" \ "\,"\text{hi there}\ ")
\]

**Q:** What is the \(\oplus\) doing in “\text{span } p\ \text{x@}(x:xs’)”?

**Higher-type goodies, 3**

\[
\text{zipWith } :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]
\]

\[
\text{zipWith'} \_ \ [\] \_ = [] \\
\text{zipWith'} \_ \ [\] \_ = [] \\
\text{zipWith'} f\ (x:xs)\ (y:ys) = f\ x\ y : \text{zipWith'} f\ xs\ ys
\]

**For example:**

\[
\text{sum } \$\ \text{zipWith } (\times)\ [2,\ 5,\ 3]\ [1.75,\ 3.45,\ 0.25] \\
\sim \quad \text{sum } [3.5,\ 17.25,\ 0.75] \\
\sim \quad 21.50
\]

\[
\text{zipWith } (\div)\ (a \rightarrow (a \times 30 + 3) \div b)\ [6,4,3,2,1]\ [1,2,3,4,5] \\
\sim \quad [153.0,61.5,31.0,15.75,6.6]
\]

**Q:** What is the \(\$\) doing??

**Q:** What is the \(\div\) doing??
Digression: The application operator

($) :: (a -> b) -> a -> b
f $ x = f x -- $ has low, right-associative binding precedence

So

$$
\text{sum} \, $ \, \text{filter} \, (> 10) \, $ \, \text{map} \, (*) \, [2..10] \\
\equiv \\
\text{sum} \, \text{filter} \, (> 10) \, (\text{map} \, (*) \, [2..10])
$$

Digression: λ-expressions

The following definitions are equivalent

\[
\text{munge}, \text{munge'} :: \text{Int} \to \text{Int} \\
\text{munge} \, x = 3 \times x + 1 \\
\text{munge'} \, x = \lambda \, x \to 3 \times x + 1
\]

So the following expressions are equivalent

\[
\text{map} \, \text{munge} \, [2..8] \\
\text{map} \, \text{munge'} \, [2..8] \\
\text{map} \, (\lambda \, x \to 3 \times x + 1) \, [2..8]
\]

So, \((\lambda \, x \to 3 \times x + 1)\) defines a “nameless” function.

We can use \((\lambda \, \_ \to \_)\) to return functional results. E.g.,

\[
\text{addNum} :: \text{Int} \to (\text{Int} \to \text{Int}) \\
\text{addNum} \, n = \lambda \, x \to (x + n)
\]

Higher-types, structural recursion on lists, 1

Consider some structural recursion on lists:

\[
\begin{align*}
\text{sum'} \, [] &= 0 \\
\text{sum'} \, (x:xs) &= x + \text{sum'} \, xs \\
\text{concat'} \, [] &= [] \\
\text{concat'} \, (xs:xss) &= xs \, ++ \, \text{concat'} \, xss \\
\text{unzip'} \, [] &= ([],[]) \\
\text{unzip'} \, ((x,y):xys) &= (x:xs,y:ys) \\
\text{where} \, (xs,ys) &= \text{unzip'} \, xys \\
\text{where} \, (xs,ys) &= (a:as,b:bs)
\end{align*}
\]

These all have the general form:

\[
\begin{align*}
\text{someFun} \, [] &= z \\
\text{someFun} \, (x:xs) &= f \, x \, (\text{someFun} \, xs)
\end{align*}
\]

So we can encapsulate this by:

\[
\begin{align*}
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
\text{foldr} \, f \, z \, [] &= z \\
\text{foldr} \, f \, z \, (x:xs) &= f \, x \, (\text{foldr} \, f \, z \, xs)
\end{align*}
\]

Higher-types, structural recursion on lists, 2

Consider some structural recursion on lists:

\[
\begin{align*}
\text{sum'} \, [] &= 0 \\
\text{sum'} \, (x:xs) &= x + \text{sum'} \, xs \\
\text{concat'} \, [] &= [] \\
\text{concat'} \, (xs:xss) &= xs \, ++ \, \text{concat'} \, xss \\
\text{unzip'} \, [] &= ([],[]) \\
\text{unzip'} \, ((x,y):xys) &= (x:xs,y:ys) \\
\text{where} \, (xs,ys) &= \text{unzip'} \, xys \\
\text{where} \, (xs,ys) &= (a:as,b:bs)
\end{align*}
\]

These all have the general form:

\[
\begin{align*}
\text{someFun} \, [] &= z \\
\text{someFun} \, (x:xs) &= f \, x \, (\text{someFun} \, xs)
\end{align*}
\]

So we can encapsulate this by:

\[
\begin{align*}
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
\text{foldr} \, f \, z \, [] &= z \\
\text{foldr} \, f \, z \, (x:xs) &= f \, x \, (\text{foldr} \, f \, z \, xs)
\end{align*}
\]

As a foldr

\[
\begin{align*}
\text{sum'} \, [] &= 0 \\
\text{sum'} \, (x:xs) &= x + \text{sum'} \, xs \\
\text{concat'} \, [] &= [] \\
\text{concat'} \, (xs:xss) &= xs \, ++ \, \text{concat'} \, xss \\
\text{unzip'} \, [] &= ([],[]) \\
\text{unzip'} \, ((x,y):xys) &= (x:xs,y:ys) \\
\text{where} \, (xs,ys) &= \text{unzip'} \, xys \\
\text{where} \, (xs,ys) &= (a:as,b:bs)
\end{align*}
\]

\[
\begin{align*}
\text{sum'} \, [] &= 0 \\
\text{sum'} \, (x:xs) &= x + \text{sum'} \, xs \\
\text{concat'} \, [] &= [] \\
\text{concat'} \, (xs:xss) &= xs \, ++ \, \text{concat'} \, xss \\
\text{unzip'} \, [] &= ([],[]) \\
\text{unzip'} \, ((x,y):xys) &= (x:xs,y:ys) \\
\text{where} \, (xs,ys) &= \text{unzip'} \, xys
\end{align*}
\]
Higher-types, structural recursion on lists, 3

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]
\[
\text{foldr} f z \; \mathsf{[]} = z
\]
\[
\text{foldr} f z \; (x:xs) = f \; x \; (\text{foldr} \; f \; z \; xs)
\]

Foldr’s cousin’s

For folds: look here
For scans: look here

Exercises

- Use \text{foldr} to define \( n \mapsto 1^2 + 2^2 + 3^2 + \cdots + n^2 \).
- Use \text{foldr} and \text{foldl} to define \text{length}.
- Use \text{foldr} and \text{foldl} to define \text{and} and \text{or}.
- Use \text{foldr} or \text{foldl} to define \text{reverse}.
- Use \text{scanr} or \text{scanl} to define \( n \mapsto [1!, 2!, 3!, \ldots, n!] \).
We can introduce a “natural number data type” by:

```haskell
data Nat = Zero | Succ Nat
```

where Zero stands for 0 and Succ stands for the function \( x \mapsto x + 1 \).

A structural recursion over Nat’s is a function of the form:

```haskell
fun :: Nat -> a
fun Zero = z
fun (Succ n) = f (fun n)
```

where \( z :: a \) and \( f :: a \to a \). So if you expand things out, you see that

\[
\text{fun } (\text{Succ } (\text{Succ } (\ldots \text{Zero}))) = (f (f (\ldots z)))
\]

where \( z \) and \( f \) are applied \( k \) many times. We can define a fold for Nat’s by:

```haskell
foldn :: (a->a) -> a -> Nat -> a
foldn f z Zero = z
foldn f z (Succ n) = f (foldn f z n)
```

Using

```haskell
data Nat = Zero | Succ Nat
foldn f z Zero = z
foldn f z (Succ n) = f (foldn f z n)
```

we can bootstrap arithmetic by:

```haskell
add m n = foldn Succ n m
times m n = foldn (‘add’ n) Zero m
```

etc.

---

**Associations**

Convention: \( \to \) associates to the right

\[
t_1 \to t_2 \to t_3 \to \cdots \to t_n \to t \equiv t_1 \to (t_2 \to (t_3 \to (\ldots (t_n \to t) \ldots))
\]

Convention: application associates to the left

\[
f x_1 x_2 x_3 \ldots x_n \equiv (\ldots (((f x_1) x_2) x_3) \ldots x_n)
\]

**WHY?**

**Suppose**

\[
f :: t_1 \to t_2 \to t_3 \to t
e1 :: t_1
e2 :: t_2
e3 :: t_3
\]

**Then**

\[
f e1 :: t_2 \to t_3 \to t
f e1 e2 :: t_3 \to t
f e1 e2 e3 :: t
\]
Currying and Uncurrying

Consider

\[
\text{comp1 :: Int -> Int -> Bool}
\]
\[
\text{comp1 x y = (x<y)}
\]

\[
\text{comp2 :: (Int,Int) -> Bool}
\]
\[
\text{comp2 (x,y) = (x<y)}
\]

Every \( f :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t \) has a corresponding \( f' :: (t_1,t_2,\ldots,t_n) \rightarrow t \) and vise versa.

In fact

\[
\text{curry2 :: ((a,b)->c) -> a -> b -> c}
\]
\[
\text{curry2 g = \x y -> g(x,y)}
\]

\[
\text{uncurry2 :: (a->b->c) -> (a,b) -> c}
\]
\[
\text{uncurry2 f = \( (x,y) \rightarrow f x y \)}
\]

Mathematically: This is just a fancier version of:

\[
(c^b)^{a} = c^{a \times b}
\]

from High School math.